图灵程序设计丛书

Jumping into C++

C++程序设计

现代方法

[美] F. Alexander Allain 著

赵守彬 陈园军 马兴旺 译



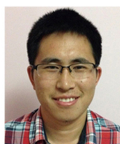
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



F. Alexander Allain

知名云存储创业公司Dropbox软件工程师、资深C++程序员，在哈佛大学工程与应用科学学院有多年一线教学经验（重点围绕C、C++、Scheme讲授计算机科学基础课程）。另外，他还是知名编程网站Cprogramming.com的创建者与维护者，上面关于C和C++的教程帮助了全球数百万的程序员。



赵守彬

2011年毕业于河北理工大学网络工程专业，后从事Android平台游戏和应用开发，擅长C/C++、Java等，目前从事基于cocos2d-x的手游开发。



陈园军

毕业于南京大学计算机系，硕士学历，微博账号“NJU陈小坏”。主要从事嵌入式系统研究，有多年的C++开发经验，关注开源社区和云计算等领域，对新技术有强烈的探索欲。他常年混迹于字幕翻译论坛，常读书，闲书、技术皆可，不求甚解，只贪欢愉，最大的爱好非科幻莫属。



马兴旺

湖南大学毕业，中国计算机学会YOCSEF委员，证通电子股份有限公司高级工程师，从事支付安全工作。

TURING 图灵程序设计丛书

Jumping into C++

C++程序设计

现代方法

[美] F. Alexander Allain 著

赵守彬 陈园军 马兴旺 译

人民邮电出版社

北 京

图书在版编目（C I P）数据

C++程序设计 ：现代方法 / (美) 阿兰
(Allain,F.A.) 著 ; 赵守彬, 陈园军, 马兴旺译. -- 北
京 : 人民邮电出版社, 2014.8
（图灵程序设计丛书）
ISBN 978-7-115-35700-7

Ⅰ. ①C… Ⅱ. ①阿… ②赵… ③陈… ④马… Ⅲ. ①
C程序—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第104270号

内 容 提 要

本书是一本写给没有编程经验的人看的 C++ 入门书，篇幅适中，通俗易懂。全书分四部分，涵盖了 C++ 编程的所有重要概念。作者在 C++ 教育领域很有影响力，他是月访问量超百万的著名 C\C++ 教程站 Cprogramming.com 的创建者，也真正了解每一位 C++ 学习者的需求，了解初学者起步阶段的困惑和纠结。因此，本书由浅入深、循序渐进、步步为营，讲述了编程过程的每一个环节，揭示了编程之路中可能遇到的各种“坑”，是初学 C++ 最合适的入门书。

本书适合 C++ 初学者、在校学生，以及对 C++ 编程感兴趣者参考阅读。

◆ 著　[美] F. Alexander Allain
译　赵守彬　陈园军　马兴旺
责任编辑　李松峰　毛倩倩
执行编辑　程　芃
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：23.75
字数：591千字　2014年 8 月第 1 版
印数：1－3 500册　2014年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2013-5709号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

目　　录

第一部分　进入 C++的世界

第三部分 编写大规模程序

第四部分 其他

Part 1 第一部分

进入 C++ 的世界

让我们准备进入程序的世界吧！编程与其他艺术形式相同，你能通过编程进行创造，但不同的是，编程可以让你的创造力因计算机的性能大大提升！你可以创造迷人的游戏，比如《魔兽世界》（*World of Warcraft*）《生化奇兵》（*Bioshock*）《战争机器》（*Gears of War*）和《质量效应》（*Mass Effect*）。你也可以创建让人身临其境的虚拟现实类游戏，比如《模拟人生》（*The Sims*）。你可以编写程序，比如浏览器（如 Chrome）、电子邮件编辑器或聊天客户端，或者是像 Facebook、亚马逊那样的网站，把人们联系在一起。你也可以满足用户的需求，为 iPhone 或者 Android 手机构建应用。当然，这都是需要花时间磨练成为高手之后才能做出来的。不过，就算你刚刚开始学习，也可以写出很多有趣的程序，比如写个程序解决你的数学作业，编写《俄罗斯方块》（*Tetris*）等小游戏在朋友面前炫耀，或者编写工具便利地解决手头上需要几天或者几周才能完成的复杂运算，等等。一旦理解了本书教给你的最基本的编程知识，你便能写出各种各样的图形或网络程序，包括游戏、科学模拟程序，等等。

C++ 是一门很强大的编程语言，它能为你学习现代编程技术打下坚实的基础。事实上，C++ 和很多编程语言原理相同，掌握 C++ 后，你能很快掌握其他编程语言（大多数编程人员都会同时掌握多种编程语言）。

C++ 程序员可以非常灵活地参与很多不同的项目。你每天用到的大部分程序和应用都是用 C++ 实现的。也许你会觉得不可思议，前面列出的每一个程序，要么完全是用 C++ 编写的，要么其重要的组件就是用 C++ 编写的[①]。

① 你可以在 http://www.stroustrup.com/applications.html 找到这些程序，以及更多的 C++ 实现。

事实上，在 Java 和 C# 如此流行的情况下，人们对 C++ 的兴趣依然在持续增长。在过去的几年里，我的网站 Cprogramming.com（http://www.cprogramming.com/）的访问量有显著增加。C++ 依然是编写高性能应用程序的首选语言，相比于用 Java 或者其他类似语言编写的程序，用 C++ 编写的程序运行速度通常更快。作为一种编程语言，C++ 一直在进步，而且有了新的语言规范 C++11，增加了很多新的特性，使开发人员可以在保证高性能的同时更容易更快速地工作[①]。深入了解 C++ 在就业上同样非常有优势，要求掌握 C++ 的工作通常具有挑战性而且薪酬很高。

你准备好了吗？本书第一部分会指导你编写程序，掌握 C++ 的基本构建方法。一旦完成此部分的学习，你便可以写出能向朋友展示的真正程序，并且学会像程序员一样思考。虽然还不能完全掌握 C++，但是你将充分准备好去学习剩下的语言特性。这些特性可以让你写出真正实用且强大的程序。

我会给出足够的背景知识和术语帮你理解 C++，等你掌握基础之后再讲解复杂的内容。

本书其他部分将逐步介绍更深入的概念：对大量数据的处理，包括从文件中获取数据；轻松高效地处理数据（并且学会使用大量的快捷键）；编写规模更大、更复杂的程序，且不会迷失在复杂的逻辑中。当然，你也会学习专业程序员所使用的工具。

通读本书，你应该能够读写真正有用而且有趣的程序。如果对游戏感兴趣，你将可以挑战真正的游戏编程。如果你正在学习或者准备学习 C++ 的相关课程，应该已经掌握了课程的基本知识。如果你在自学，应该可以使用所有 C++ 提供的工具去编写感兴趣的任何程序了。

勘误与更新

虽然我竭尽全力保证本书内容的准确性和时新性，但是本书中所用到的一些工具却会不断有新版本出现。因而，本书中的一些内容（包括代码）可能面临过时，或者错误的可能。因此，我专门为此提供一个地方供大家查看更新与勘误：http://www.cprogramming.com/errata.html[②]。

致谢

感谢 Alexandra Hoffer 细致、耐心的编辑工作，感谢她在本书出版过程中提供的宝贵建议。因为有了她的帮助，本书才得以出版。另外，感谢 Andrei Cheremskoy、Minyang jiang 和 Johannes Peter 给出的反馈、建议，感谢他们的斧正。

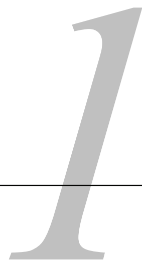
---

① 本书编写几近结束时该规范才刚刚试行，所以我没有引用新的标准。你可以在 http://www.cprogramming.com/c++11/what-is-c++0x.html 找到一系列关于 C++11 的介绍。

② 中文版的勘误查询及提交地址：ituring.cn/book/1263。读者也可以在同一个页面下载本书示例代码的源文件。
——编者注

第 1 章

简介和环境搭建



1.1 什么是编程语言

如果想控制计算机，你需要一种可以和计算机对话的方法。不像猫或狗那样有一套自己的神秘语言，计算机的语言是人类创造的。计算机程序是一段文本，就像一本书或一篇文章，有着特定的结构。编程语言对人类来说易于理解，但与自然语言相比，它们的结构更严格，词汇量也更小。C++便是这些计算机语言中很流行的一个。

在你写完程序之后，计算机需要一种方法来运行它，就是解释所写的是什么，也就是执行程序。执行的过程取决于所使用的编程语言和开发环境，我们稍后会讲到如何执行程序。

目前有很多种编程语言，尽管每种语言都有自己的语法和关键字，但它们在很多方面是相通的，一旦学会一种，再学习其他的便会容易很多。

1.2 C 和 C++之间的不同之处

C语言是为了开发Unix操作系统而诞生的，是一种低层且强大的语言，但缺少很多现代编程特性。C++是基于C语言的一个较新的语言，它增加了很多现代编程语言特性，比C更加易用。

C++包含了所有C语言的强大特性，同时提供了很多新功能，使其更容易编写复杂程序。

例如，C++可以非常方便地管理内存。它有很多的特性可以实现“面向对象编程”和“泛型编程”，我们之后会解释这些是什么意思。目前你只需要知道C++对程序员来说更加地简单，可以让他们无需考虑机器运行的细节，而是将注意力放在如何解决问题上。

如果你在学C还是学C++之间徘徊，我强烈建议学习C++。

1.3 学习 C++之前，是否需要先了解 C

不需要。C++是C的超集，任何能用C做的事情，都可以用C++完成。如果了解C语言，你会

很快适应C++的面向对象特性。如果不了解C，也不要紧，先学会C并不会有多少优势，其实你能很快掌握并运用C++的独有特性（比如更简单的输入和输出）。

1.4　成为程序员，是否需要懂数学

如果每次有人问我这个问题我都收他5分钱的话，那我这笔财富要用计算器才能算清。很幸运，这答案是：不需要！大多数编程涉及的是设计和逻辑推理，而不是快速运算、线性代数或微积分。数学和编程之间的重叠部分，主要集中在逻辑推理和严密的思维部分。只有需要编写高级3D图形引擎（http://www.cprogramming.com/tutorial.html#3dtutorial）、数控编程，或是写程序来进行统计分析时，你才需要真正的数学知识。

1.5　术语

本书中我会不断定义新术语，不过会从一些非常基本的概念开始介绍。

1.5.1　编程

编程指的是编写计算机能够理解和执行的指令，这些指令称为*源代码*。我们将在接下来的内容中初步接触一些源代码。

1.5.2　可执行文件

编程的最终成果就是生成一个*可执行文件*。可执行文件即计算机能够运行的文件：如果使用Windows系统，这些文件即EXE文件。Microsoft Word便是一个可执行文件。有些程序会有额外的文件（图形文件、音乐文件等），但每个程序都必须有一个可执行文件。为了生成可执行文件，你需要一个将程序源代码转化为可执行文件的*编译器*。没有编译器，除了眼巴巴地看着源代码，你无法做任何事情。这实在是太无聊了，我们赶紧安装一个编译器吧。

1.6　编辑和编译源文件

本章接下来的部分讲述如何搭建一个简单易用的编程环境。我推荐安装两个工具：一个编译器和一个*编辑器*。你已经知道编译器的存在是为了让程序可以工作。虽然之前没有提及编辑器，但它非常重要，编辑器可以让你按照正确的格式编辑源代码。

源代码必须以*纯文本*的形式编写。纯文本文件仅包含文字信息，不包含格式化信息。用Microsoft Word（或类似产品）创建的文件就不是纯文本文件，因为它包含字体、字号等格式化

信息。在Word中打开文件时，尽管你看不到这些信息，但它们确实存在。而纯文本文件只有原始信息，可以使用接下来我们将要介绍的工具来创建和编辑。

编辑器还会提供两个很方便的功能：*语法高亮*和*自动缩进*。语法高亮给代码添加颜色，使你可以很容易地区分程序的不同元素。自动缩进可以自动对代码排版，使其更加易读。

如果你使用Windows或者Mac，我推荐更复杂的编辑器：包含了编辑器和编译器的*集成开发环境*（Integrated Development Environment，IDE）。如果你使用Linux，我推荐使用非常简单易用的编辑器nano。下文中，我将指导大家进行设置。

1.7 关于示例源代码

本书包含大量示例源代码，所有源代码都可以供你自由使用；使用无任何限制，但不保证质量。本书附带的示例源代码（sample_code.zip[①]）按章划分文件夹（例如，本章对应文件夹ch1）。本书中的每段源代码都有对应的相同名称的文件（非章名）。

1.8 Windows

我们在Windows下安装的工具是Code::Blocks，一个免费的C++开发环境。

1.8.1 第 1 步：下载Code::Blocks

- 打开网址http://www.codeblocks.org/downloads；
- 单击链接“Download the binary release”，访问http://www.codeblocks.org/downloads/26；
- 进入Windows 2000/XP/Vista/7/8部分；
- 查找名字中含有mingw的文件（本书翻译之时，其名称为codeblocks-13.12mingw-setup.exe；你看到的版本号可能会与此不同）；
- 保存文件到桌面（本书翻译之时，这一文件大约为97.86 MB）。

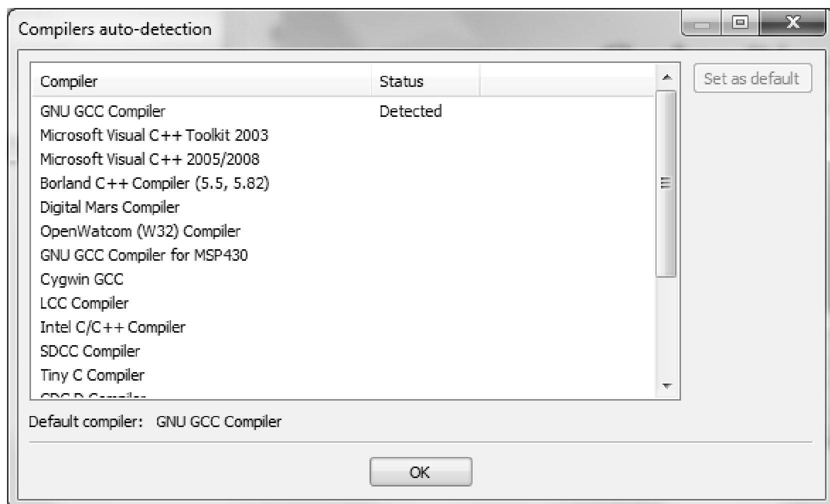
1.8.2 第 2 步：安装Code::Blocks

- 双击安装文件；
- 连续单击**Next**，一些教程会假定你安装在**C:\Program Files\CodeBlocks**（默认安装路径），但其实可以选择安装在其他位置；
- 进行完整安装（在下拉菜单Select the type of install中选择Full: All plugins, all tools, just everything）；
- 运行Code::Blocks。

① 读者可在图灵社区iTuring.cn本书页面免费注册下载。——编者注

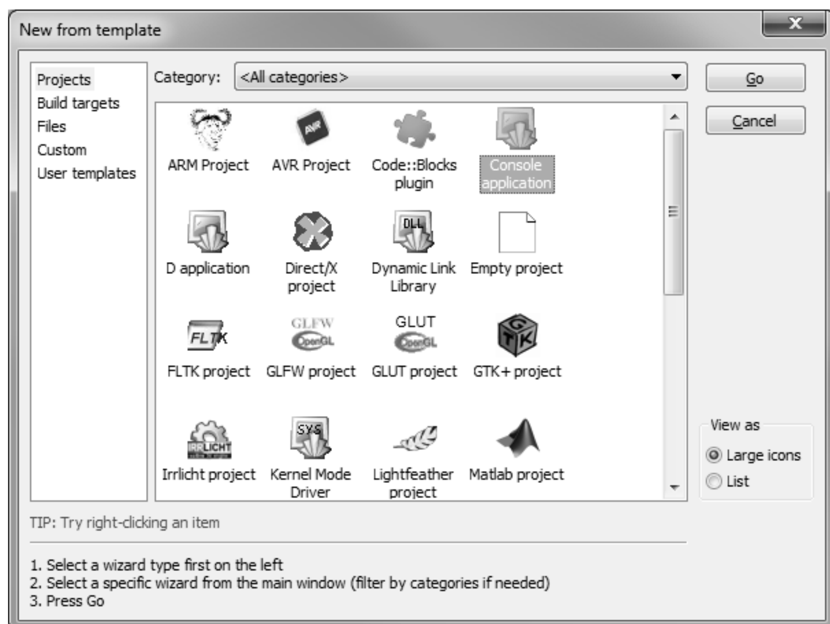
1.8.3 第 3 步：运行Code::Blocks

系统会弹出Compilers auto-detection（编译器自动检测）窗口：



如果出现编译器自动检测窗口，单击OK按钮即可。Code::Blocks可能会询问你是否关联C/C++文件，建议进行关联。单击File按钮，在New选项下，选择Project...。

接下来会出现如下窗口：



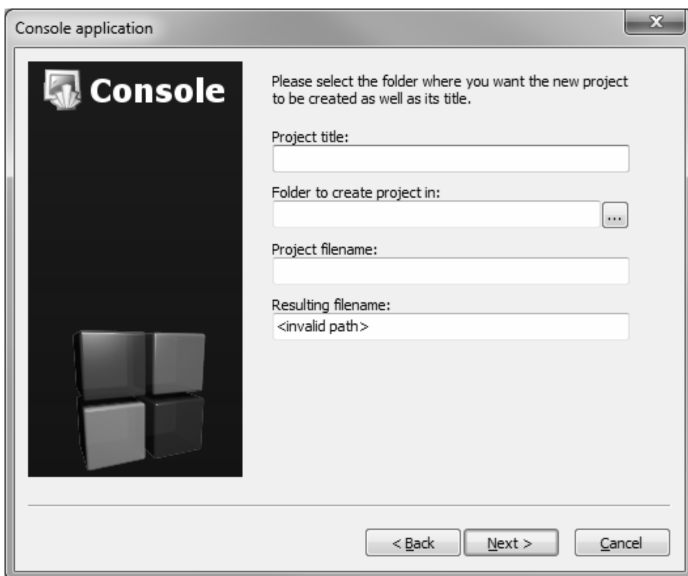
选择Console Application选项，然后单击Go按钮。书中所有的示例代码都以控制台程序运行。

连续单击Next直到出现语言选择对话框：



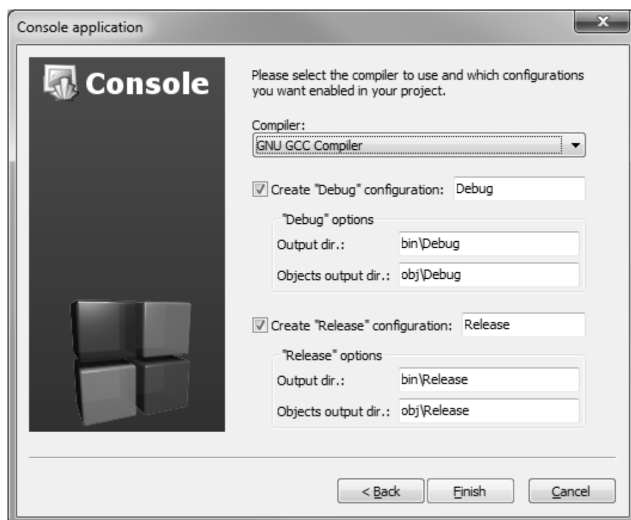
对话框让你选择C或C++，这里学的是C++，所以选择C++。

继续单击Next，Code::Blocks会提示你选择程序保存路径：



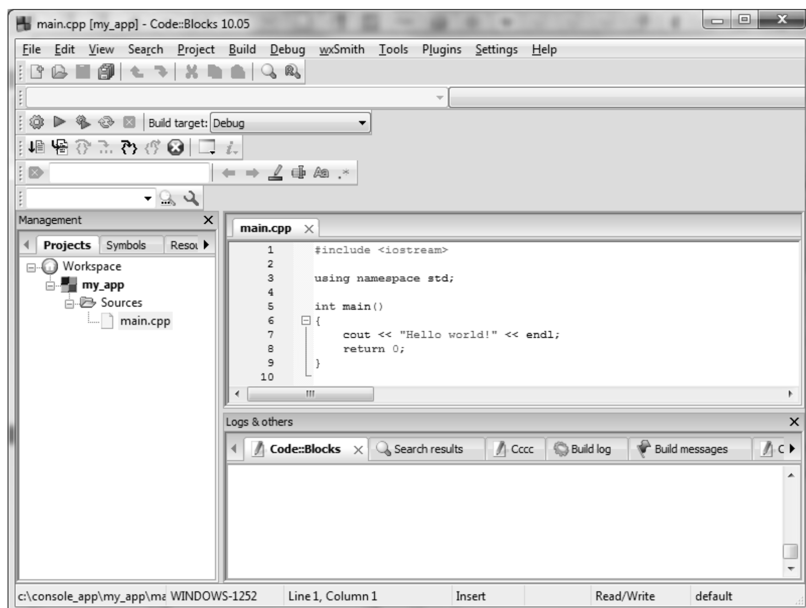
我推荐将其保存在单独的文件夹里面，因为Code::Blocks可能会创建很多文件（尤其是创建其他类型的项目时）。你还要给项目起一个名字，任意名字都可以。

再次单击Next，程序会提示你选择编译器：



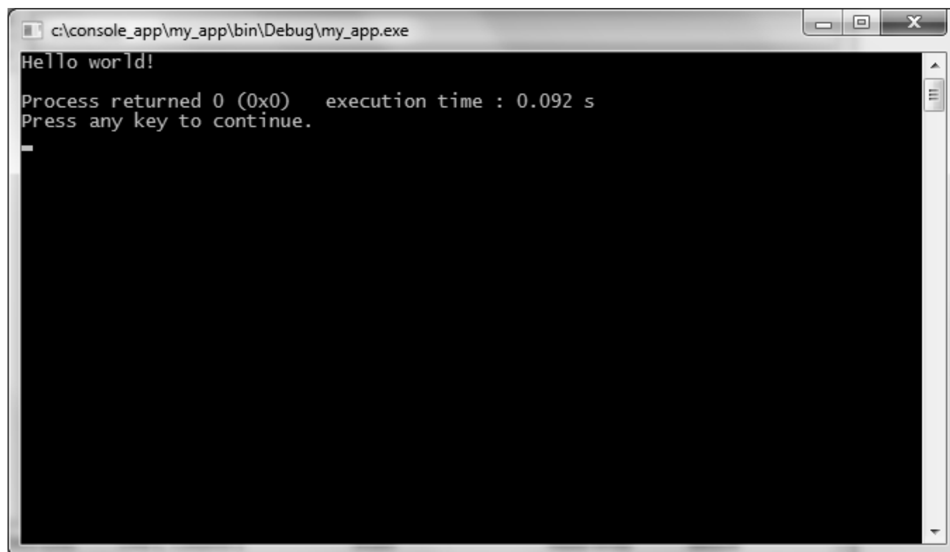
这里不需要做任何操作，保留默认值即可，然后单击Finish按钮。

现在，你可以打开左边的main.cpp文件了。



（如果找不到main.cpp，你可能需要展开Sources文件夹。）

至此，你有了自己的main.cpp文件，且可以随意修改它。注意文件的扩展名是.cpp而不是.txt，.cpp是C++源文件的标准扩展名，尽管C++源文件就是纯文本。目前它只能输出“Hello word !”。我们来运行它吧。单击F9，编译，然后运行。（你也可以选择Build | Build and Run选项。）



现在已经成功地运行了一个程序！你可以简单修改一下main.cpp，按F9键再次编译和运行。

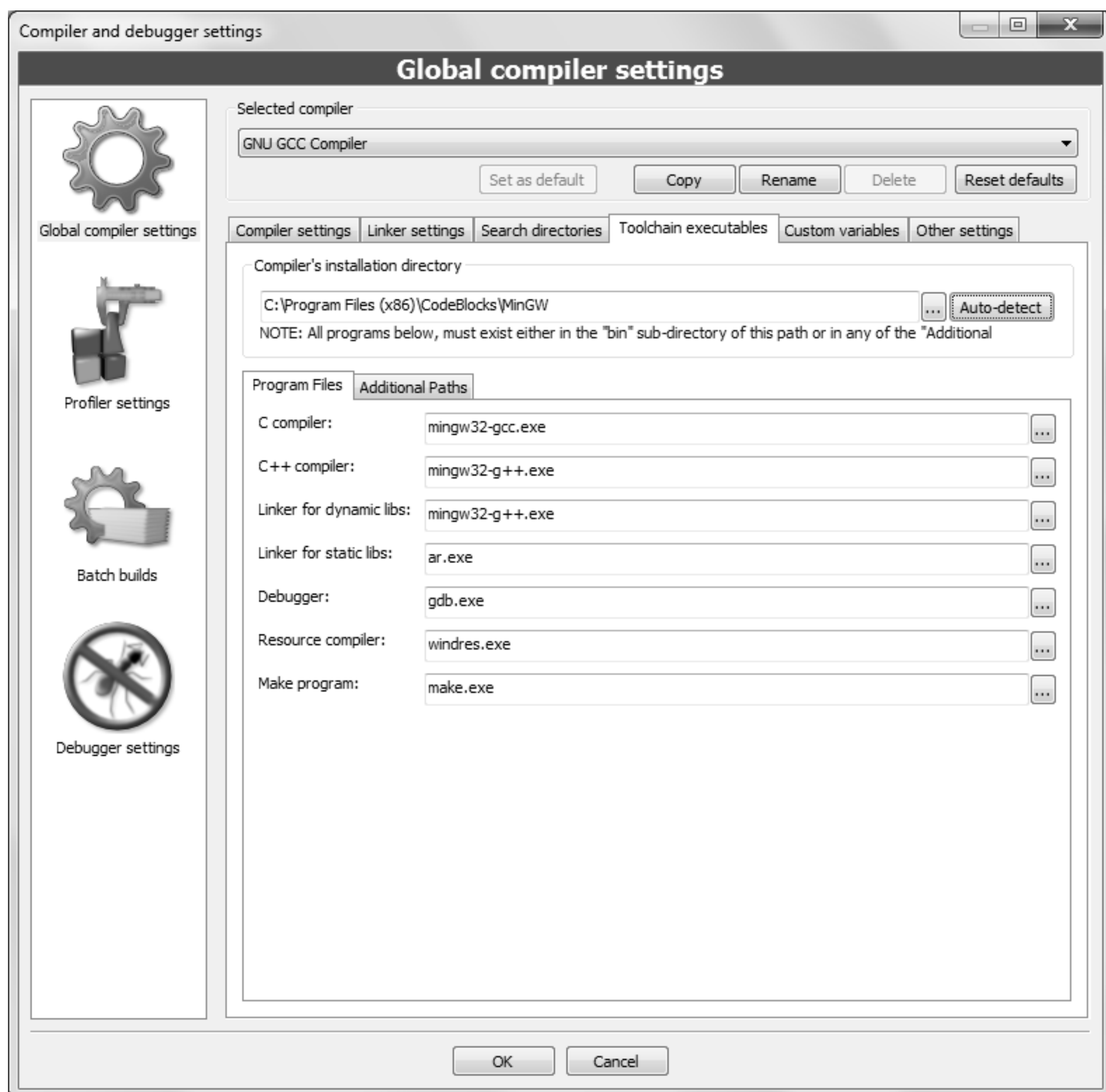
1.8.4　错误调试

如果无法运行程序，很可能是因为编译错误或者编译环境没有配置好。

1. 环境设置

运行故障时最常见的错误是与此类似的消息：“CB01-Debug” uses an invalid compiler. Probably the toolchain path within the compiler options is not setup correctly?! Skipping...。

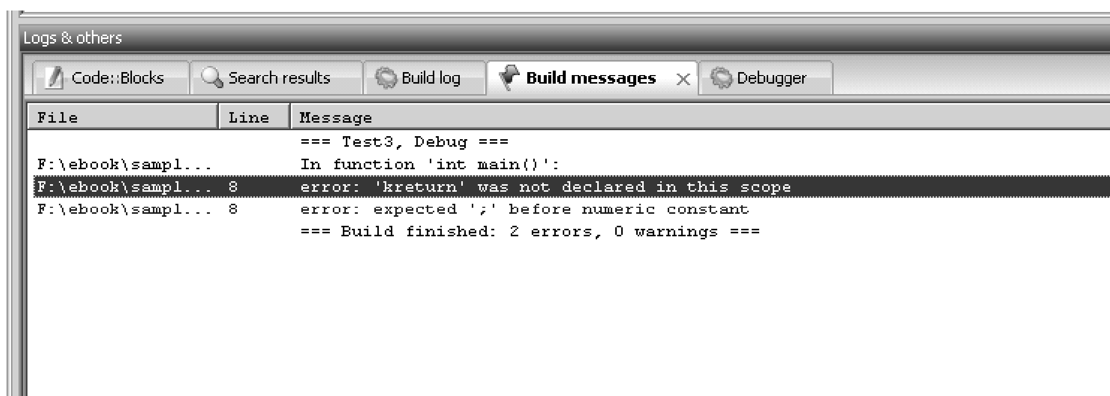
首先，确保下载的Code::Blocks是包含MinGW的完整版本；如果问题仍没有解决，很可能是编译器自动检测出了问题。接下来检查auto-detected的状态，找到Settings | Compiler and Debugger...，选择左边的Global Compiler Settings（有一个齿轮状图标），然后选择右侧的Toolchain executables选项卡，选项卡上有一个Auto-detect按钮，单击它应该能够解决问题。如果仍未解决，你需要手动填写表单。下图是我的系统配置的演示截图，请更改Compiler’s installation directory为你自己的实际路径（如果安装在了其他某个位置），并且确保所有内容填写如下图。



完成修改后，按F9，看看能否正常运行程序。

2. 编译错误

如果你修改了main.cpp但编译器无法识别，则说明可能发生了编译错误。想找出错误原因，可以查看Build messages或Build log窗口。Build messages窗口仅显示编译错误，Build log则会显示其他信息。下面显示了一个编译错误：



从示例中可以看出，错误信息会给出文件名、代码行号和简短的错误描述。在这段代码中，我把`return 0 ;`改成了`kreturn 0 ;`，它不是C++的有效语法，所以出错了。

编程过程中遇到编译失败时，通过这个窗口可以获取有用的信息。

在本书中，你会看到大量示例代码。对于每个示例代码，你都能够参考创建一个新的控制台程序，或者直接修改附带的源文件。我建议创建新的控制台程序，以便修改示例代码并将其保存留待以后查看。

1.8.5 使用Code::Blocks的原因

我在书的开头提到了集成开发环境，Code::Blocks便是一个集成开发环境，编码和编译都可以用它轻松搞定。不过需要注意一下，Code::Blocks本身不是编译器。你下载Code::Blocks时，安装包里面包含了一个编译器，本书中编译器是MinGW（http://www.mingw.org/）的GCC，它是Windows下的一个免费编译器。Code::Blocks已经帮你处理了所有安装和调用编译器的工作。

1.9 Macintosh

本节主要讨论OS X系统上的环境设置。[①]

OS X自带了一个非常强大的基于Unix的shell环境，所以本书在1.10节介绍的大部分工具也可以在OS X下使用。当然，你可能想尝试苹果的Xcode集成开发环境。不管是不是想用Xcode，使用标准Linux工具前必须安装Xcode。

① 如果你使用Mac OS 9或更早的系统版本，在无法升级的情况下，可以试用Macintosh Programmer’s Workshop，官网链接是http://developer.apple.com/tools/mpw-tools/。OS 9之前的系统太古老了，这里不再介绍安装步骤。

在Mac下开发C++程序并不需要使用Xcode。只有你想开发Mac图形程序时才需要学习使用Xcode。

1.9.1　Xcode

Xcode是Mac OS X自带的一个免费软件，但默认情况下不会安装。你可以在Mac OS X的光盘中找到安装文件，也可以在网上下载最新版本。包含文档的下载版本体积非常大，如果网速比较慢，建议先尝试在光盘中找安装文件。注意，像gcc、g++这些基本的编译器，Linux通常会默认安装，但Mac OS X不会，想使用它们，必须下载Xcode Developer Tools。

注意，目前Xcode有Xcode 5和Xcode 6 beta两个版本。下文包含了Xcode 5和Xcode 6 beta两个版本的安装说明。[①]

1.9.2　安装 Xcode 5

你可以直接在Mac App Store中搜索Xcode 5。下载完成后，Dock上会出现一个Install Xcode图标，单击就可以进入安装过程。

安装过程要求同意许可协议，然后它列出需要安装的组件；选择默认的组件即可。安装时请全部选择默认选项，直到安装完成。

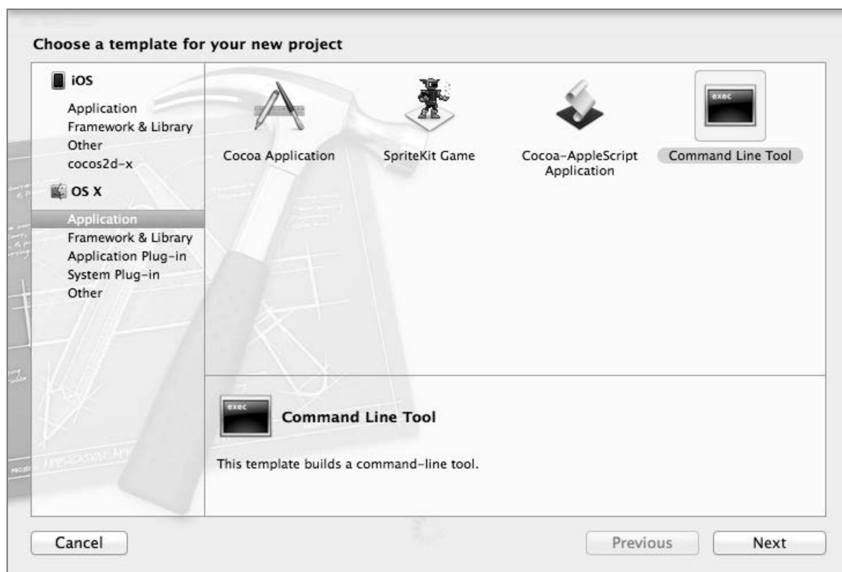
1.9.3　运行 Xcode

安装完成后，你可以在Developer|Applications|Xcode下找到Xcode；单击运行。Xcode附带大量文档，你需要花费一些时间学习Xcode Quick Start Guide教程；可以通过单击开始界面上的Learn about using Xcode链接看到。但接下来的学习过程假设你没阅读过任何其他文档。

1.9.4　用 Xcode 创建第一个 C++程序

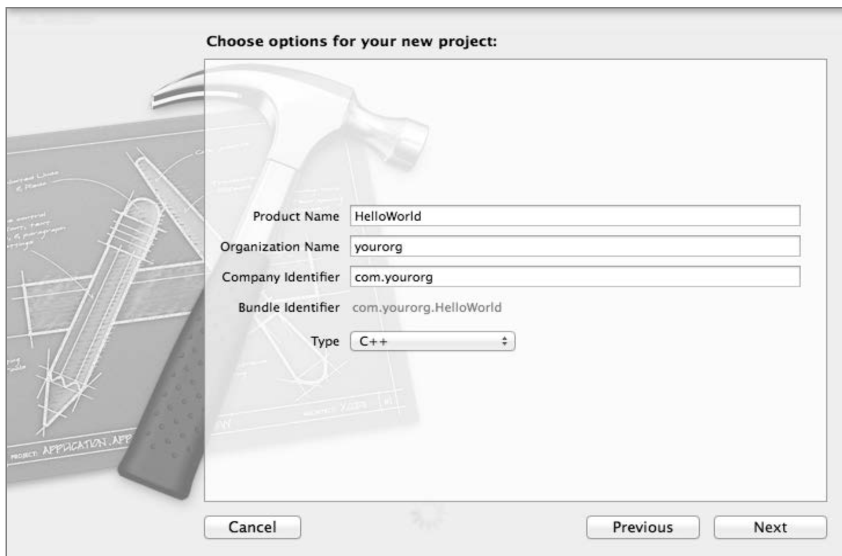
运行Xcode，在主界面上选择Create a new Xcode project（也可以选择File|New|New Project…，或者按快捷键Shift-⌘-N）。此时会出现如下界面：

① 本书英文版撰写之时，Xcode有Xcode 3和Xcode 4两个版本。但本书中文版即将出版时已有Xcode 5和Xcode 6 beta两个版本，本节后续内容已更新到这两个版本。——编者注



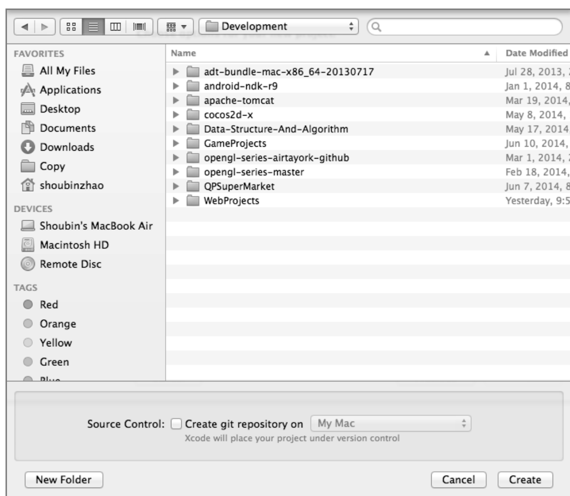
选择在左边栏Mac OS X下方的Application，然后选择Command Line Tool。（你可能也会看到iOS下方的Application，不过请暂且忽略它。）然后单击Next。

之后会看到如下界面：



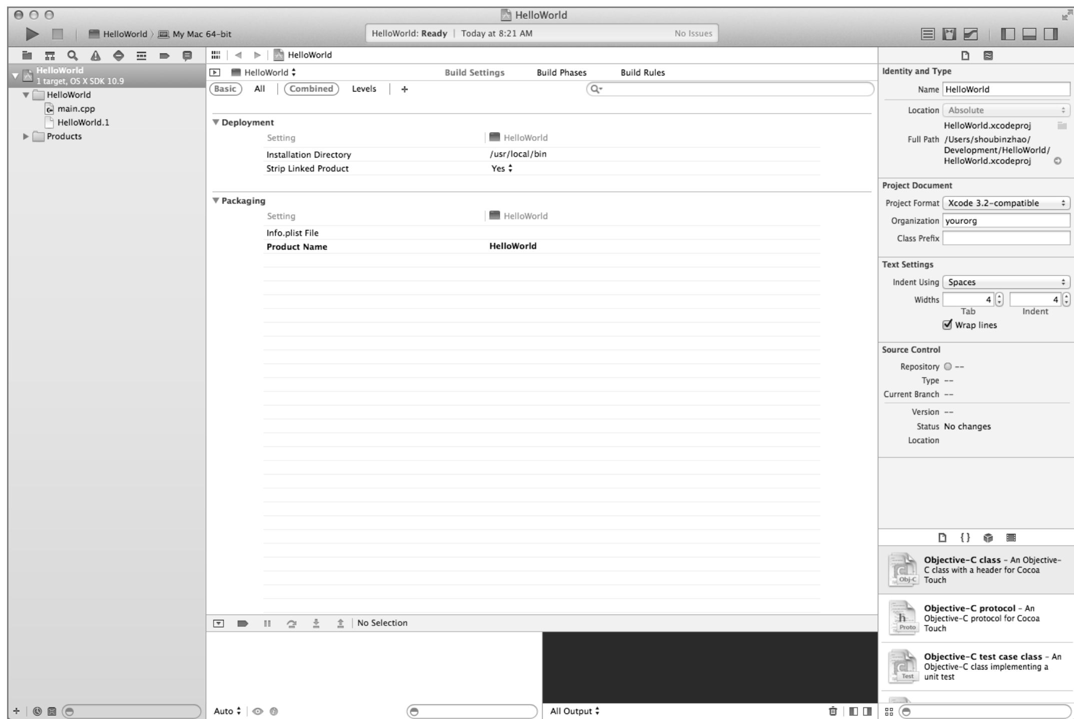
将项目名HelloWorld填到输入框里，选择Type为C++（默认可能为C），然后单击Next按钮。

单击之后，会出现如下界面：



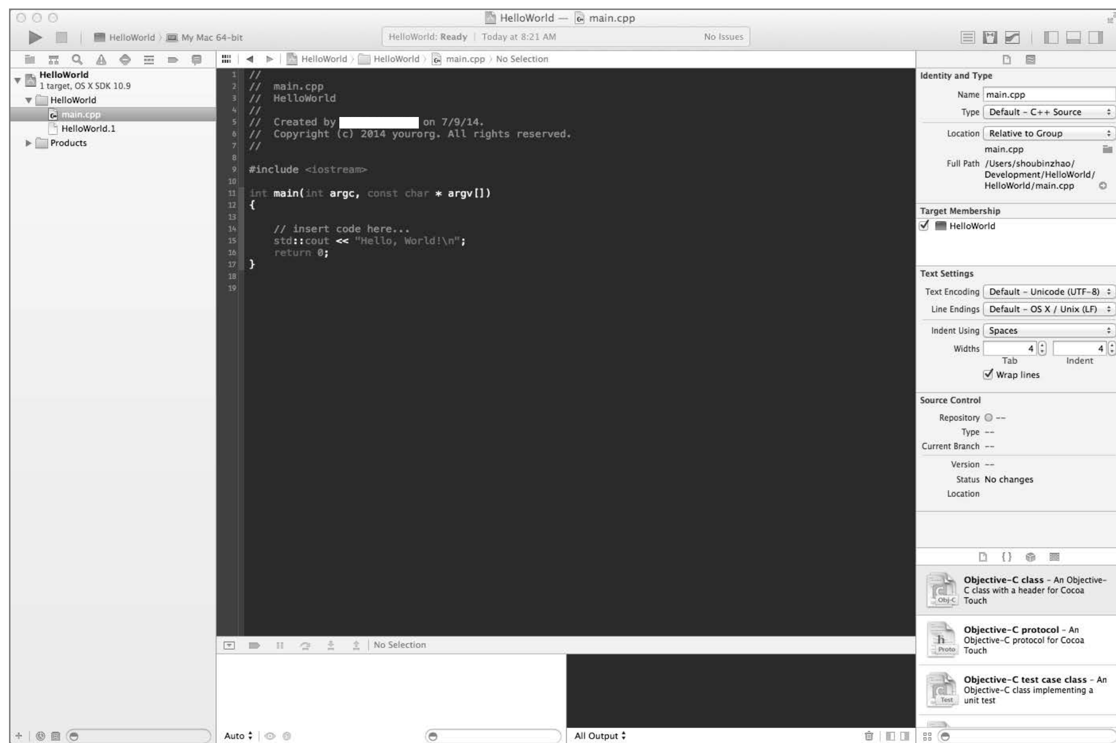
如果Create git repository on被选中，就反选它。Git是一个版本控制系统，会保留项目的多个版本，但Git超出了本书的讲述范围（所以先取消其选中状态）。然后，选择项目的存放位置——我把它放在Documentation目录中。完成这些后，请单击Create。

单击之后，会出现一个新的窗口，如图所示：



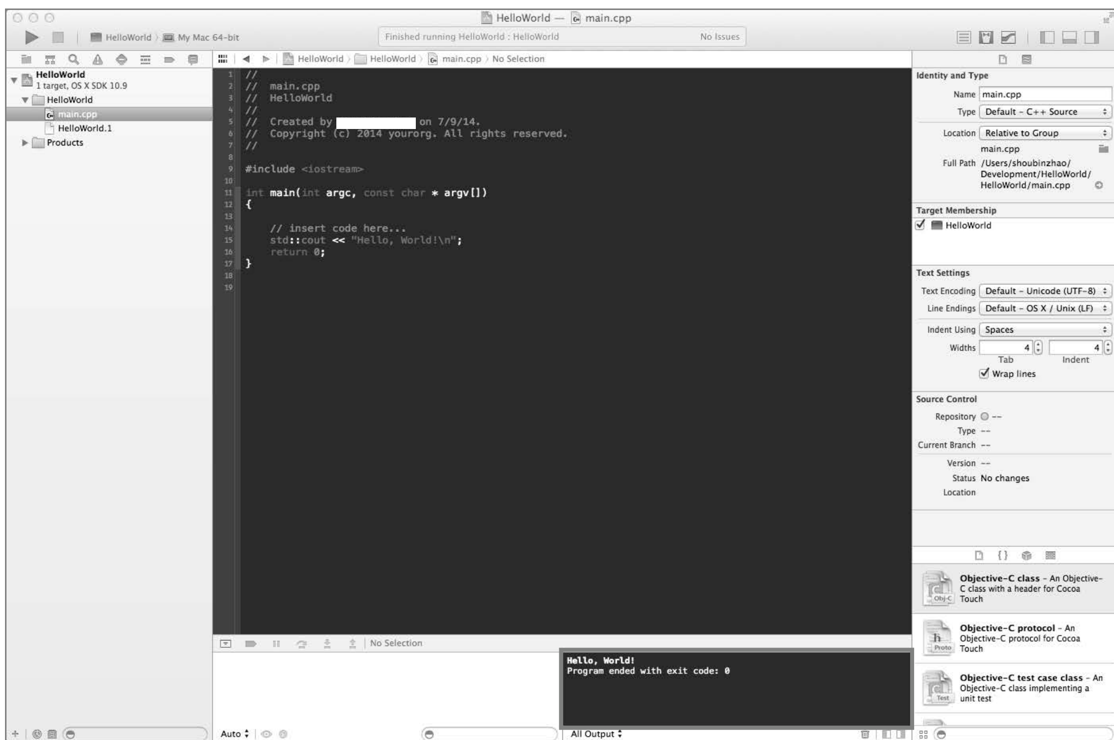
窗口中包含了很多信息。左侧边栏包含了源代码和产品。源代码放在与项目名同名的文件夹下，比如示例中的HelloWorld文件夹。窗口中剩下的区域会显示编译器信息，现在我们不需要对其做任何操作。

至此，让我们开始编辑源代码。从视图中左侧边栏中选择main.cpp（注意，文件的扩展名是.cpp而不是.txt，.cpp是C++的标准扩展名，尽管cpp文件就是纯文本）。单击，源代码便会显示在主窗口上。现在你可以直接进行修改了。



你也可以双击文件，然后会看到可以移动的编辑窗口。

Xcode默认提供了一个小示例程序。让我们来编译和运行。单击工具栏上的Run按钮，视图右下角就会输出相关信息（见视图下方被框起部分）。



就这样，你成功运行了第一个应用程序！

从现在开始，当你想运行示例程序时，可以使用我们刚刚创建的项目，也可以新建项目然后编译执行。如果你想增加自己的代码，可以从修改Xcode自带的示例程序中的main.cpp开始。

1.9.5　安装Xcode 6 beta

你可以直接在Mac App Store中搜索Xcode 6 beta。下载完成后，Dock上会出现一个Install Xcode图标，单击就可以进入安装过程。

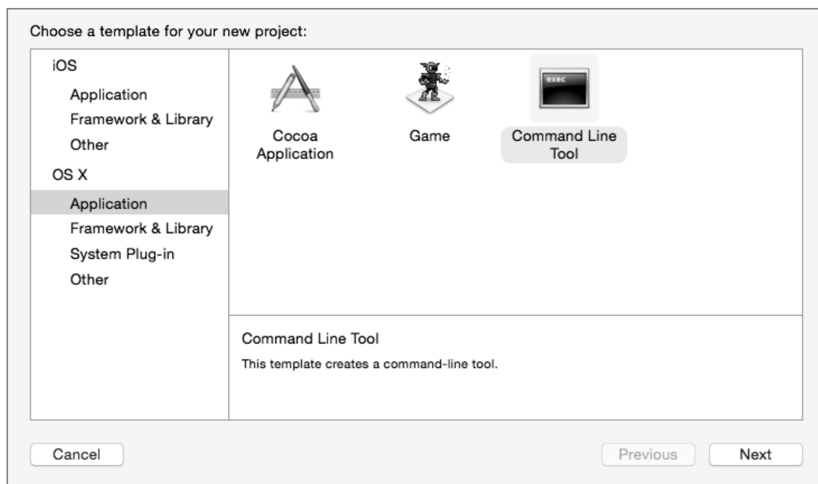
安装过程要求同意许可协议，然后它列出需要安装的组件；选择默认的组件即可。安装时请全部选择默认选项，直到安装完成。

1.9.6　运行Xcode

安装完成后，你可以在Developer|Applications|Xcode下找到Xcode；单击运行。Xcode附带大量文档，你需要花费一些时间学习Xcode Quick Start Guide教程；可以通过单击开始界面上的Learn about using Xcode链接看到。但接下来的学习过程假设你没阅读过任何其他文档。

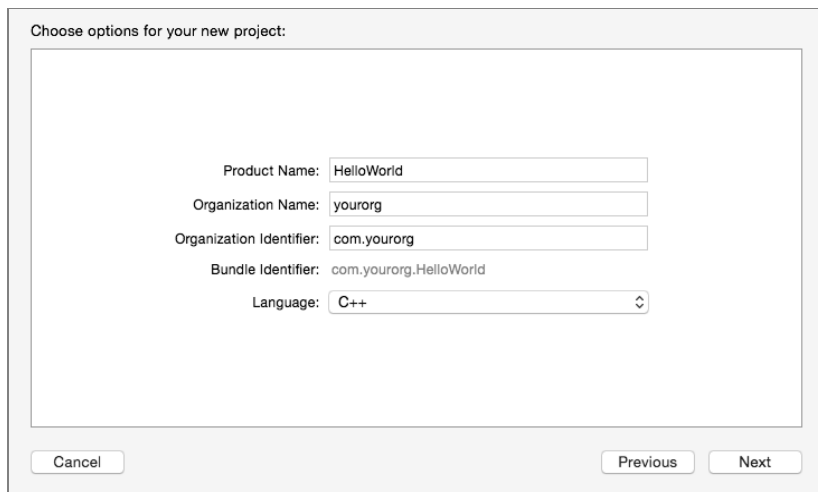
1.9.7 用Xcode创建第一个C++程序

运行Xcode，在主界面上选择Create a new Xcode project（也可以选择File|New|New Project...，或者按快捷键Shift-⌘-N）。此时会出现如下界面：



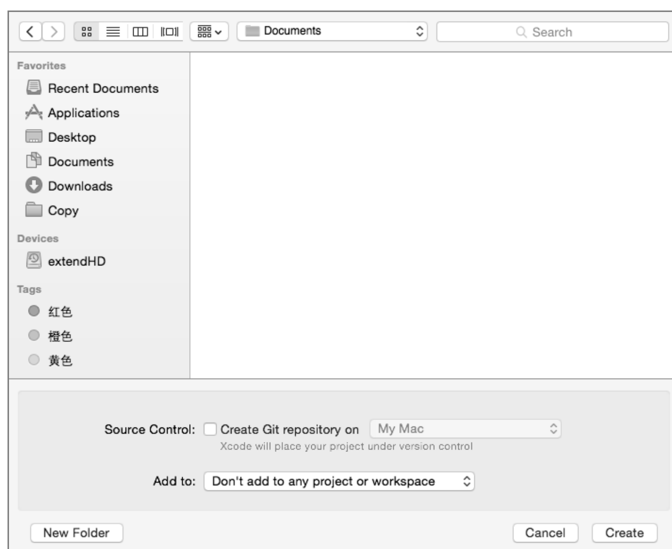
选择在左边栏Mac OS X下方的Application，然后选择Command Line Tool。（你可能也会看到iOS下方的Application，不过请暂且忽略它。）然后单击Next。

之后会看到如下界面：



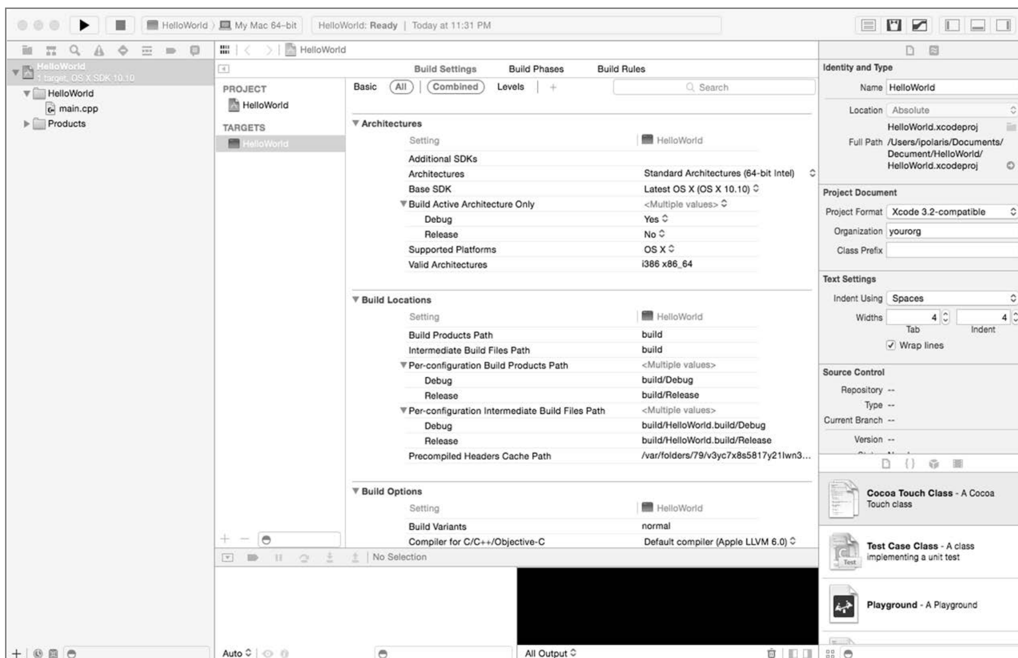
将项目名HelloWorld填到输入框里，选择Type为C++（默认可能为C），然后单击Next按钮。

单击之后，会出现如下界面：



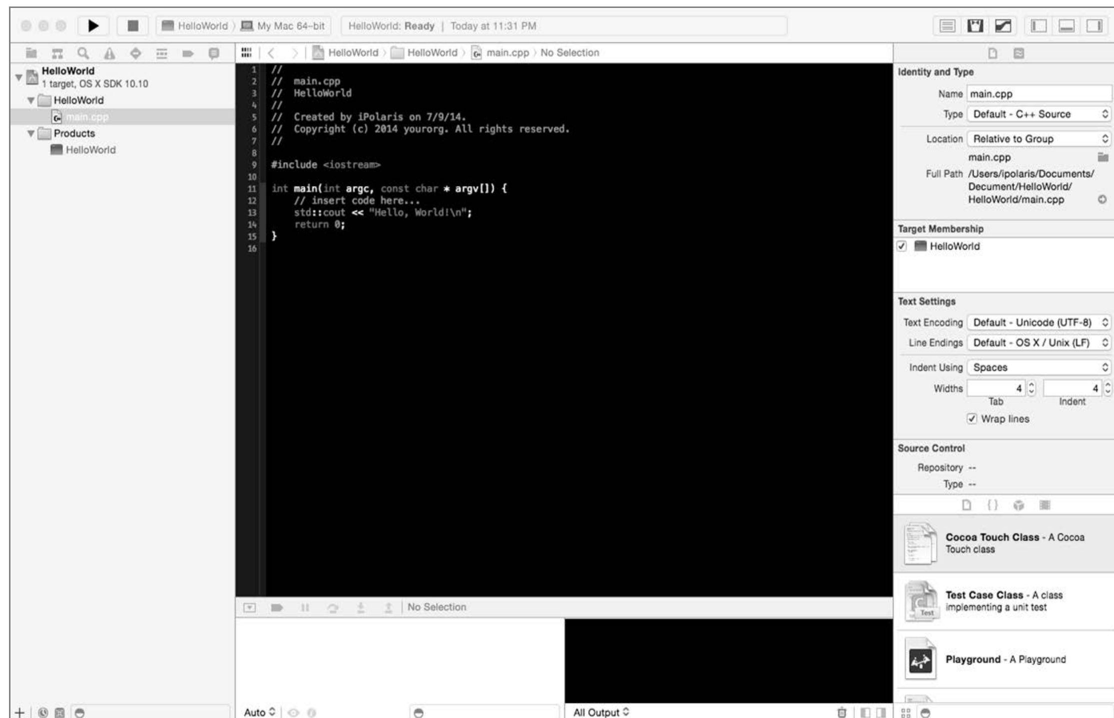
如果Create Git repository on被选中，就反选它。Git是一个版本控制系统，会保留项目的多个版本，但Git超出了本书的讲述范围（所以先取消其选中状态）。然后选择项目的存放位置——我把它放在Documentation目录中。完成这些后，单击Create。

单击之后，会出现一个新的窗口，如图所示：



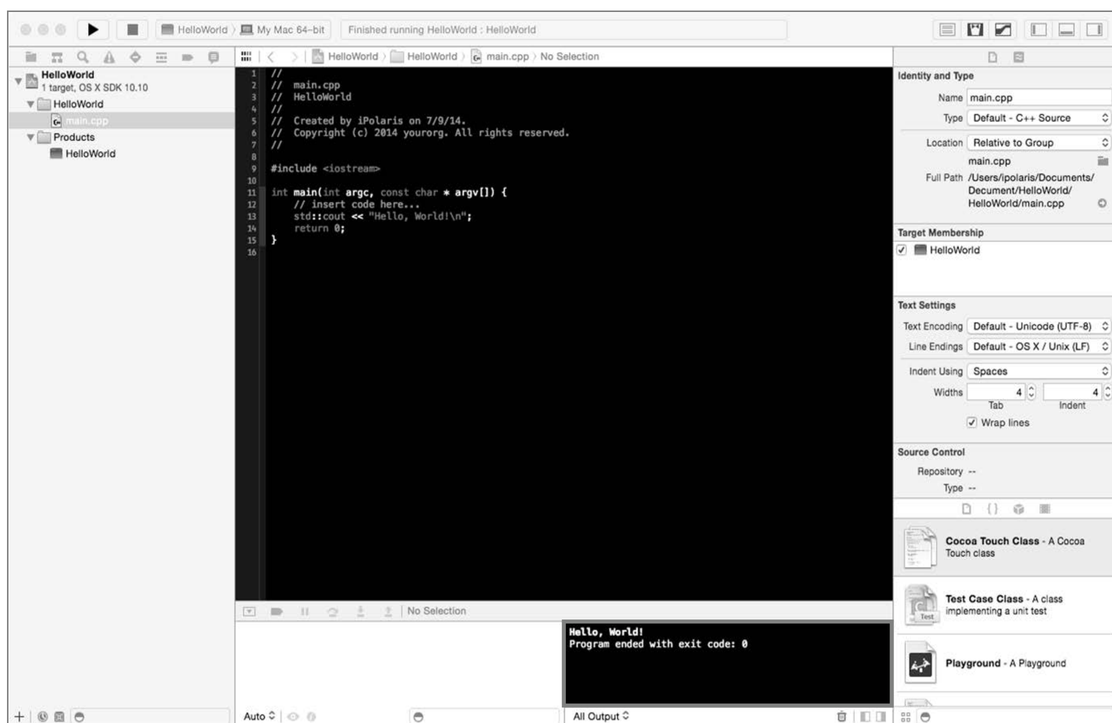
窗口中包含了很多信息。左侧边栏包含了源代码和产品。源代码放在与项目名同名的文件夹下，比如示例中的HelloWorld文件夹。窗口中剩下的区域会显示编译器信息，现在我们不需要对其做任何操作。

至此，让我们开始编辑源代码。从视图中左侧边栏中选择main.cpp（注意，文件的扩展名是.cpp而不是.txt，.cpp是C++的标准扩展名，尽管cpp文件就是纯文本）。单击，源代码便会显示在主窗口上。现在你可以直接进行修改了。



你也可以双击文件，然后会看到可以移动的编辑窗口。

Xcode默认提供了一个小示例程序。让我们来编译和运行。单击工具栏上的Run按钮，视图右下角就会输出相关信息（见视图下方被框起部分）。



就这样，你成功运行了第一个应用程序！

从现在开始，当你想运行示例程序时，可以使用我们刚刚创建的项目，也可以新建项目然后编译执行。如果你想增加自己的代码，可以从修改Xcode自带的示例程序中的main.cpp开始。

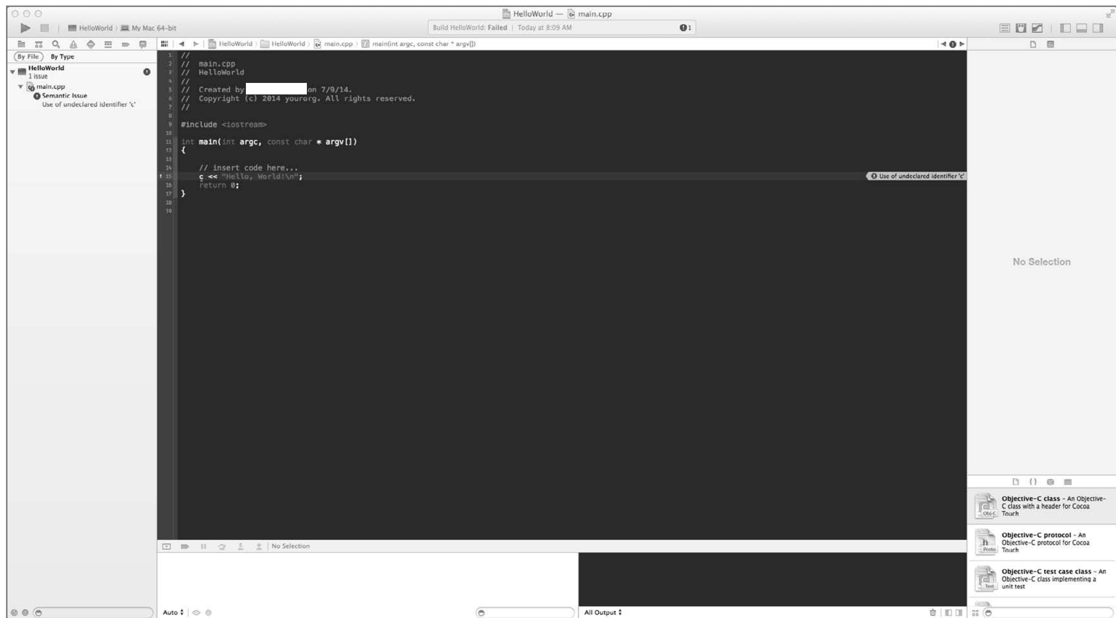
1.9.8　错误调试①

程序可能会因为一些原因编译失败，通常是因为编译错误。（例如，示例代码可能打字错误，也可能程序中存在真正的错误。）如果出现编译错误，编译器会输出一个或多个编译错误信息。

Xcode直接在出错代码行旁显示编译错误信息。在下面的示例程序中，我把原来程序中的输出函数`std::cout`修改成单个字符`c`（见下页图）。

如构建失败截图所示，构建失败和编译失败信息显示在正上面的条形框中，点击右侧的叹号，左侧边栏中会出现具体的错误信息。

① 本节使用的是Xcode 5的截图。Xcode 6和Xcode 5基本相似。



修改完错误，你只要单击Build and Run按钮就可以重新调试。

1.10　Linux

如果你使用Linux，应该已经安装了C++编译器。一般来说，Linux用户使用的C++编译器是g++；g++是GNU Compiler Collection（GCC）的一部分。

1.10.1　步骤 1：安装g++

打开终端窗口，输入g++，按回车键。如果已经安装编译器，就会出现：

```
g++: no input files
```

如果你看到这样的信息：

```
command not found
```

那么需要安装g++，而安装g++取决于使用的Linux发行版的包管理软件。如果使用Ubuntu，只需要输入：

```
aptitude install g++
```

其他的Linux版本可能包含同样简单的包管理软件，也可能需要其他额外的操作步骤。请阅读你

使用的Linux发行版的帮助文档，以获取更多的信息。

1.10.2　步骤 2：运行g++

运行g++相对比较简单。首先，我们要创建第一个程序。创建一个扩展名为.cpp的简单文件，它包含的文本如下：

```
#include <iostream>

int main ()
{
      std::cout << "Hello, world" << std::endl;
}
```

示例代码1：hello.cpp

保存文件为hello.cpp，并记住文件保存的路径。（注意，文件扩展名是.cpp而不是.txt，.cpp是C++的标准扩展名，尽管cpp文件就是纯文本。）

返回终端窗口，进入你保存文件的目录。输入下列内容然后按下回车键：

```
g++ hello.cpp -o hello
```

g++的参数-o选项表示输出文件的名称。如果没有使用-o，名字默认为a.out。

1.10.3　步骤 3：运行你的程序

本例中，程序的名字为hello，所以需要输入下面的命令来运行程序：

```
./hello
```

这样应该就可以看到输出了：

```
Hello, world
```

这是你的第一个程序，向美好的新世界打个招呼吧。

错误调试

程序可能会因为某些原因编译失败，通常是因为编译错误（例如，示例代码可能打印错误，也可能程序中存在真正的错误）。如果出现编译错误，编译器会输出一个或多个编译错误信息。

例如，如果在示例程序中的cout前加上一个x，那么编译器会返回如下错误：

```
gcc_ex2.cc: In function 'int main ()':
gcc_ex2.cc:5: error: 'xcout' is not a member of 'std'
```

每个错误都会显示文件名、行号和错误信息。示例中，问题出在编译器无法理解xcout上，因为它应该是cout。

1.10.4 步骤 4：安装文本编辑器

如果你使用Linux，会想使用一个好的文本编辑器。Linux下有很多非常高端的文本编辑器，如Vim（http://www.vim.org/）和Emacs（http://www.gnu.org/software/emacs/）。（我在Linux环境下工作时使用Vim。）不过它们入门比较难，需要你投入大量时间。从长远看，花时间学习是值得的，但刚开始学习编程时，可能不想花这么长时间去学习使用编辑器。如果你已经熟悉两个工具中的任一个，那么就继续使用吧。

如果你还没有钟爱的编辑器，可以试一下nano。nano（http://www.nano-editor.org/）是一个相对简单的文本编辑器，但它包含非常有用的语法高亮和自动缩进功能（这样你就不用一直在换行的时候按tab键了；听起来无所谓，但非常需要）。nano基于pico编辑器；pico非常易用，但缺少很多编程时需要的功能。如果你用过邮箱程序pine，那么可能使用过pico；没用过也不要紧，使用nano无需任何经验。

如果已经安装了nano，在终端窗口中输入nano即可，它会自动启动。如果没有安装，并且出现以下信息：

```
command not found
```

此时你需要安装nano，可以按照Linux的软件包管理器获取软件的方式获取nano。本文采用的是2.2.4版本的nano，不过，之后的版本也可以。

1.10.5 配置nano

为了能够使用nano，需要安装一个nano配置文件。其配置文件名为 .nanorsc，和大部分Linux配置文件一样，你的个人配置文件放在home文件夹下（~/.nanorc）。

如果文件已经存在，只需要简单地编辑一下；否则，你需要创建它。（若你没有任何使用Linux文本编辑器的经验，可以使用nano进行配置；请阅读下文，如果你需要了解nano的基本知识。）

使用本书附带的示例.nanorc文件可以正确配置nano。它提供了漂亮的语法高亮和自动缩进，让编码变的更加简单。

1.10.6 使用nano

如果想新建一个文件，可以不带参数运行nano。你也可以在命令行里指定一个文件名来编辑那个文件：

```
nano hello.cpp
```

若文件不存在，nano将会在内存中创建一个文件，直到你进行保存，才会在硬盘上保存创建的文件。

下图是nano运行时的示例：



上方的矩形框里面显示的是正在编辑的文件的文件名；如果在启动软件时没有提供文件，则显示New Buffer。

下方的矩形框里面是一堆键盘命令。如果看到字母面前有^，说明你需要将字母和Ctrl键一起按下。比如，“退出”的键盘命令是^X，所以必须要同时按下Ctrl和X；不区分大小写。

如果你一直使用Windows，可能不熟悉nano的术语，因此让我们先学习一下基本的nano操作。

1. 文本编辑

你可以启动nano新建一个文件或者打开一个存在的文件。此时已经可以输入内容了，在这点上nano和Windows上的Notepad非常相似。但如果想用复制和粘贴，它们就不相同了，nano的剪切键是Ctrl-K，复制键是Ctrl-U。若你没有选择任何文本，命令会默认剪切一行。

查找文本时使用Ctrl-W，然后会出现很多选项，最简单的方法就是输入要查找的内容然后按回车键。

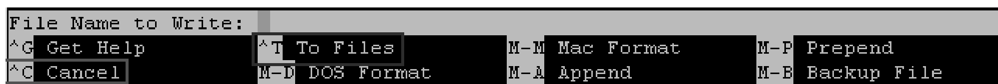
使用Ctrl-Y可跳转到前一页，使用Ctrl-V可跳转到后一页。注意，其快捷键操作和Windows中几乎完全不同。

nano唯一缺少，而其他大部分编辑器都包含的重要功能是被默认禁止的撤销/重做功能，nano目前（在版本2.2中）只是实验性地支持撤销/重做。nano默认禁用撤销/重做功能。

使用Alt-R可以进行文件范围内的替换/查找：首先提示输入需要查找的文本，然后提示输入替换文本。

2. 保存文件

在nano中，保存文件的术语是写入（WriteOut，快捷键是Ctrl-O）。



当你调用写入功能，软件提示要输入文件名，即便文件已经打开。如果你在编辑已存在的文件，文件名默认显示，可以直接按回车保存文件。如果想保存在新的位置，可以输入新的文件名进行保存，或者进入文件选择菜单To Files（Ctrl-T）选择文件保存目录。取消（Ctrl-C）是针对命令本身的——大部分命令都有取消本身操作的选项，和Windows不同的是，默认的取消键是Ctrl-C而不是 Esc。目前无须理会其他命令，一般情况下我们使用不到它们。

3. 打开文件

如果想打开一个文件进行编辑，需要用到读取文档（Ctrl-R），读取文档会出现下面的菜单选项：



假设你想打开一个文件，而不是在编辑的文本中插入内容。在选择文件前选择菜单中的New Buffer；New Buffer的快捷键是M-F，M指的是元键，在示例中，你通常使用键盘中的Alt键：Alt-F①，告诉nano去打开文件。执行完后，你可以输入文件的名称，或者用Ctrl-T打开文件列表进行选择。操作过程中同样可以采用Ctrl-C取消文件选择操作。

4. 查看源代码

你已经初步了解了如何使用nano，现在已经可以打开源文件并开始编辑了。如果.nanorc文件配置正确，当打开具有某些文本的源文件时，代码会根据函数的不同来标记不同的颜色，比如我们之前编辑的源代码hello.cpp，会如下图所示，其中“Hello, World”呈现粉红色。

① 有些人使用Alt键时可能没有反应，你可以先单击一次Esc键，然后再按字母键；比如，Alt-F可以使用Esc F代替。

```
server3.cprogramming.com - PuTTY
  GNU nano 2.2.4              File: /home/cprogram/.hello.cpp

#include <iostream>

int main()
{
        std::cout << "Hello, world" << std::endl;
}


                              [ Wrote 7 lines ]
^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text    ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text  ^T To Spell
```

语法高亮功能依赖文件扩展名，只有你保存源文件为.cpp，它才会出现高亮。

从现在起，当你需要运行示例程序时，可以按照上面的步骤，先用nano创建新的文本文件，然后进行编译。

5. 知识拓展

现在你已经知道如何在nano中编辑基本文件，如果想了解更多，可以使用Ctrl-G调用内置的简易帮助。下面的网址中包含nano的大量高级功能：http://freethegnu.wordpress.com/2007/06/23/nano-shortcuts-syntax-highlight-and-nanorc-config-file-pt1/。

第 2 章

C++基础

2.1 C++简介

如果你按照前面一章的描述搭建好了开发环境，应该已经成功运行了自己的第一个程序。恭喜！这是个良好的开端。

本章，我们来学习C++的基本模块，让你能够独立编写简单的程序。下面我将介绍几个你会反复遇到的概念：程序结构、main函数、标准函数、程序注释，以及如何像程序员一样思考。

2.1.1 最简单的C++程序

让我们从最简单的程序（不实现任何功能）入手，循序渐进地学习。

```
int main ()
{
}
```

示例代码2：empty.cpp

看，多么简单！

第一行代码：

```
int main ()
```

告诉编译器有一个叫main的函数，这个函数返回一个整数（注意：在C++中，整数简写为int）。函数是人们编写的一段代码，代码里通常调用其他函数或语言的基础函数。此例中，函数内部没有任何操作，不过我们很快就会编写包含某些操作的函数。

main函数比较特殊，它是C++程序中唯一的必须包含的函数。main函数指向程序运行的起始点。使用时我们需要在main之前指明它的返回值，如int。当一个函数有返回值时，调用该函数的代码能够获取到函数的返回值。在main函数的例子中，返回值传递给操作系统。通常这里需要一个明确的返回值，但C++允许main函数省略返回声明，默认返回0（通知操作系统程序运行正常）。

花括号“{”和“}”分别是函数开始和结束的标志（很快我们会看到其他的代码块）。你可以认为它们代表开始和结尾。此例中函数没做任何事情，因为两个花括号之间没有内容。

当运行这个简单的程序时，你看不到任何输出。因此，让我们往代码里面添加一点有趣的东西（只是一点点）。

```
#include <iostream>

using namespace std;
int main ()
{
    cout << "HEY, you, I'm alive! Oh, and Hello World!\n";
}
```

示例代码3：hello.cpp

首先，注意花括号之间有了内容——这意味程序会执行某些操作！让我们按部就班地分析这个程序。

第一行：

```
#include <iostream>
```

是一个include语句，它告诉编译器在生成可执行文件前把isotream头文件放到这段程序中。（iostream头文件由编译器提供，以提供输入输出功能。）使用#include可以将头文件中所有的内容加入到程序中。通过引用头文件，你可以调用编译器提供的众多函数。

我们需要引用包含基础函数的头文件来调用基础函数。iostream头文件包含了需要使用的大部分基础函数，并且几乎所有的程序都以这个头文件开头。此外，大部分程序都会包含一个或多个这样的引用声明。

紧跟着include语句的是这行：

```
using namespace std;
```

这是大部分C++程序都会包含的样板代码，初学时只要把它放在include语句之后、程序的开头即可。这段代码能让你很方便地调用iostream头文件中例程的缩写。我们之后会详细讲述它的工作方式，现在只需要记得包含它就可以了。

请注意行末的分号。分号是C++语法的一部分，它告诉编译器语句在这里结束，C++里大部分语句用分号终止。很多新手会忘记使用分号，因此当程序无法正常编译时请确保没有遗漏分号。当我介绍一个新的概念时，会告诉你是否需要使用分号。

接下来便是main函数，程序从这里开始执行：

```
int main ()
```

程序的下一行有一个奇怪的符号<<：

```
cout << "HEY, you, I'm alive! Oh, and Hello World!\n";
```

C++使用cout对象（读音为“C out”）输出文本，iostream头文件中包含了cout函数，而这就是我们引用 iostream头文件的原因。

<<符号称为“插入运算符”（insertion operator），程序用它来指明输出内容。简言之，cout<<实现了将文本作为参数传递给函数的函数调用，函数调用就是运行与函数相关的代码。调用函数时通常需要使用参数，此例中我们提供的字符串便是参数。参数等同于等式中的系数，比如计算正方形面积的公式是边长的平方，这里的边长便相当于函数的参数。和公式类似，函数采用变量作为参数。此例中函数把参数输出到屏幕上。

引号的作用是让编译器输出特殊符号除外的字符串原文。\n符号是特殊符号的一种，它的功能是换行，效果与按下键盘上的回车键一样，即将光标移到下一行（稍后对此做详细介绍）。有时候，你会遇到用特殊值endl作为换行的情况：cout << "Hello" << endl和cout << "Hello \n"本质上相同。①

再次强调一下分号，调用函数时需要在末尾加上它。

最后用一个花括号结束函数，这个程序便可以编译和运行了。你可以直接打开随书附带的源文件进行编译和运行，或者自己敲一遍代码。当然，也可以直接复制粘贴，不过建议你自己敲一遍代码，代码并不多，还可以帮你熟悉编译器相关细节，如分号的使用。

当第一个程序跑起来后，为什么不尝试修改cout函数来练习C++编程呢？试试输出不同的内容或输出多行文本——看看你能让计算机做些什么。

2.1.2　程序无法运行的原因

当你运行本书附带的程序时，可能看不到结果——程序一闪而过然后关闭。这种情况的发生取决于使用的操作系统和编译器，如果使用本书推荐的环境就不会遇到这个问题，如果是其他的环境就可能遇到。你可以在程序结尾增加下述代码解决这个问题。

```
cin.get();
```

这行代码会让程序在结束前等待你按键输入一个值，所以可以在窗口关闭前看到程序运行的结果。

① 单词“endl”代表“end line”，最后的字母是“L”，不是数字“1”。把“L”当成“1”写成“end1”是一个常见的错误，要谨慎。——译者注

2.1.3　C++程序的基本结构

瞧，这么短的程序有这么多的知识点。让我们剥离所有细节，看看C++程序的基本框架。

```
[include statements]
using namespace std;

int main()
{
      [你的代码写在这里];
}
```

如果上面的某一段被删掉会出现什么情况？

如果删掉include语句或者using namespace std，程序将无法编译。如果程序无法编译，说明其中有些地方编译器无法理解——可能是语法错误（比如少了一个分号）或者头文件丢失。在刚开始编程时，追踪编译错误会比较困难，任何编译失败都会产生一个或多个编译错误，这些编译错误会提示错误原因。下面是一个常见的编译错误。

```
error: 'cout' was not declared in this scope
```

如果错误信息中出现以上内容，请确保代码包含了iostream头文件并声明了using namespace std;。

编译错误有时候难以理解。如果删除了分号，可能会得到各种各样的编译错误——通常错误会出现在丢失分号的那行代码的后面，因此如果看到一大串莫名其妙的错误，请查看上一行是否有分号。随着时间的推移，你会越来越擅长分析编译错误，并且编译错误会越来越少。因此在刚开始时，遇到一堆编译错误不要觉得很糟糕，这是学会解决错误的必经之路。

2.2　为程序添加注释

在学习编程的同时也应该学习如何为代码添加说明（如果没有其他人阅读，就为你自己添加）。这个过程就是给代码添加注释。在接下来的学习中，我会非常频繁地使用注释帮助解释示例代码。

当你告诉编译器一段文本是注释时，编译器便会忽略这段文本，任何用来描述代码的文本都可以作为注释。添加注释时可以使用 //，告诉编译器这行剩下的部分是注释，也可以使用/*和*/，它们中间所有被隔断的文本都是注释。

```
//这是一行注释
此行代码不是注释

/*这是一个多行注释
此行是注释的一部分
*/
```

某些编译环境会改变注释区域的颜色，表示这段文本不是可执行代码。这也是语法高亮的一个例子。

当学习编程时，通过注释掉（comment out）一部分不想进行编译的代码来观察输出的改变，是一个非常有用的方法。例如，如果想观察程序没有cout语句时的现象，就可以把cout语句注释掉。

```
#include <iostream>

using namespace std;

int main ()
{
//     cout <<"HEY, you, I'm alive! Oh, and Hello World!\n";
}
```

示例代码4：hello_comment.cpp

要注意，不要意外注释掉其他有用代码。

如果有效代码被注释掉了，比如注释掉了头文件，程序便可能无法正常编译。如果编译代码时出现很多错误，你可以尝试注释掉可能不正确的代码，如果注释之后程序能够编译，那么问题就出在这段被注释掉的代码中。

2.3　像程序员一样思考，创建可复用的代码

让我们暂且远离编程的语法，花点时间来谈谈编程的经验。曾经有一个联邦农业保险公司的广告，其中洗车公司将汽车交还给顾客时未将洗车时用的肥皂水冲洗干净[①]。而某些保险公司不会因此赔偿顾客，因为洗车公司完成了它的职责——洗车，服务中未说明要清理洗车后的肥皂水，尽管它理应被包含。

这个广告也是对如何思考编程的一个完美比喻。计算机好比广告中的洗车公司，非常条理化且不理解隐含的要求，会精确执行你告诉它们要做的事情。如果你说“洗车”，它们就会“洗车”。如果你想要“冲洗”，就得继续说“冲洗”。一开始把要求详细落实到每一个步骤可能会让你气馁，但它能确保不遗失任何一个步骤。

幸运的是，编程时，一旦你告诉计算机如何处理某些事情，就可以进行命名和引用它，不需要一遍遍地重复这些步骤。没有想象中那么枯燥？是的，你只需要写一次精确的指令，然后引用它们即可。当我们讲解到函数时，你会很快了解这种机制。

---

① 你可以在下面地址看到，广告只有57秒：http://www.youtube.com/watch?v=QaTx1J7ZeLY。

2.4 痛并快乐着的练习

尽管你才刚刚学习这本书，不过本章结尾还是安排了几道练习题。在我看来，实践是学习编程的最好方式。首先，编程需要注重细节，从经验中我知道很容易在阅读一段代码后想“嗯，它们都行得通”，但这种想法也恰恰说明我并未掌握其中细节。不自己动手编写代码，你就无法真正掌握C++语法和语言的细微差异，初学编程时，人们很难想到可以简化程序的好方法。本书的大部分章中都准备了练习题；数量不多，我极力推荐你学完一章后，在开始学习下一章前尝试完成所有的练习题。

恭喜！你已经了解了第一个C++程序，并且懂得了一点儿程序员的思考方式。你可以多调试示例代码，看看能做些什么。下一章，我们将会探讨如何与用户交互，学习如何把用户输入放到程序中。

2.5 问答题

(1) 程序正确执行后，会返回给操作系统什么值？

A. `-1` B. `1` C. `0` D. 程序不返回值

(2) 所有C++必须包含的函数是？

A. `start()` B. `system()` C. `main()` D. `program()`

(3) 什么符号用在代码段的开始和结尾？

A. `{ }` B. `->` 和 `<-`
C. `BEGIN` 和 `END` D. `(` 和 `)`

(4) 大部分C++程序以什么符号结尾？

A. `.` B. `;` C. `:` D. `'`

(5) 下面哪个是正确的注释符号？

A. `*/ Comments */` B. `** Comment **`
C. `/* Comment */` D. `{ Comment }`

(6) 使用`cout`需要包含哪个头文件？

A. `stream` B. 不需要包含，它默认可用
C. `iostream` D. `using namespace std;`

2.6 实践题

(1) 编写一个能输出你名字的程序。

(2) 编写一个程序，在屏幕上显示多行文本，每一行显示一个你朋友的名字。

(3) 尝试注释掉我们所编程序中的每一行，观察程序能否编译。这些编译错误代表什么？你能找出程序改变后出现这些变化的原因吗？

第3章

用户交互和变量

到目前为止，你已经学习了如何编写简单的程序来显示输入的信息，学会了如何为程序添加注释。这棒极了！但如果想和用户进行交互该怎么办呢？

与用户进行交互，你需要接受外部信息的输入。要做到这一点必须对输入进行存储。在编程中，将输入的数据以及其他的数据存储在变量中。不同类型的信息（例如数字和字母）存储在不同的变量中；当声明一个变量时，必须包括数据类型以及变量的名称。

最常见的基本数据类型有char、int和double。一个char型的变量能存储一个字符，int型的变量能存储整数（不包含小数的数字），double变量可以存储包含小数的数字。（名字很奇怪是吗？）这些变量类型是声明变量时使用的关键词。

3.1 变量

3.1.1 C++中的变量声明

你只有先声明变量，才能使用变量（编译器对提前告之的事情很挑剔）。使用语法“type <name>;”声明变量。（请再次注意分号！）

下面是声明变量的例子：

```
int whole_number;
char letter;
double number_with_decimals;
```

同种类型的变量可以在同一行声明，变量间用逗号隔开。

```
int a, b, c, d;
```

我推荐一行只声明一个变量，这样容易阅读。

3.1.2 使用变量

你已经知道了如何让编译器识别变量，那么如何来使用它们呢？

使用cin（读音“C in”）来接受输入，后面跟着反方向的插入操作符“>>”，之后便是你想让用户输入的变量。

下面是一个演示如何使用变量的简单程序。

```
#include <iostream>

using namespace std;

int main ()
{
    int thisisanumber;
    cout << "Please enter a number: ";
    cin >> thisisanumber;
    cout << "You entered: " << thisisanumber << "\n";
}
```

示例代码5：readnum.cpp

让我们逐行分解并测试这个程序。第一部分你已经看过了，所以我们主要分析main函数。

```
int thisisanumber;
```

这行声明thisisanumber为整型。接下来一行是：

```
cin >> thisisanumber;
```

函数cin >> 把用户输入的值用thisisanumber存储起来。用户输入之后必须按回车键，程序才会读取数据。

3.1.3 程序闪退的处理方法

如果你之前使用cin.get()来阻止程序闪退，即使使用cin.get()上面的程序在运行时可能依然会闪退。你可以在cin.get();前增加cin.ignore();来解决这个问题。

cin.ignore()函数会读取并丢弃一个字符，此例中将读取并丢掉用户按下的回车键。当用户向程序输入字符时，回车键也被接收，但我们并不需要，所以应当丢弃。只有当你使用cin.get()让程序等待用户输入时才会用到这个函数，若没有这行，cin.get()会读取换行符，程序依然会闪退。

记住当变量被声明为整数时，若用户输入小数，小数部分将会被截断（数字的小数部分将会被忽略，比如3.141 5会变成3）。运行示例程序时，请试着输入小数或字符串。不同的输入会有不同的反应，无论你输入什么它都能够正常响应。正常的程序需要进行错误处理，不过目前我们不需要关心这些。

```
cout << "You entered: " << thisisanumber << "\n";
```

这行代码用于输出用户的输入。注意变量没有引号。如果用引号把thisisanumber引起来，程序将会输出“You Entered: thisisanumber.”。没有引号时编译器会把thisisanumber识别成变

量，程序会检查变量的值，将变量名替换成该变量的赋值然后将结果输出。

顺便提一句，不要被一行中有两个插入操作符弄晕了，一行中包含多个插入操作符是完全可行的，并且所有的输出都会被输出在同一个地方。你必须用插入操作符（<<）将字符串常量和变量分开，用一个<<同时输出字符串常量和变量会出错：

错误代码

```
cout << "You entered: " thisisanumber;
```

像调用其他函数一样，行末是一个分号。如果忘记分号，编译时会出现编译错误。

3.1.4 修改、使用和比较变量

读入和输出变量很快会让人觉得没意思。接下来让我们修改变量，让程序根据变量的不同赋值给予不同的回应。很快，我们就可以以不同的方式回应用户的不同输入。

你可以使用赋值操作符=将值传递给变量：

```
int x;

x = 5;
```

设置x等于5。你可能会认为等号会对左右两边的值进行比较，但这里等号不是比较。在C++中，用来判断等式的是由两个等号组成的的操作符==。==经常用在if语句或循环语句中。接下来的几章里，我们会学习如何根据用户的不同输入采取不同的计算，过程中会用到大量的比较操作。

```
a == 5 // 不是把5赋值给a，而是检查a是否等于5
```

你也可以对变量执行算术运算。

*	两个值相乘
-	两个值相减
+	两个值相加
/	一个值除以另一个值

下面是几个示例：

```
a = 4 * 6; // (注意分号和注释的使用) a等于24
a = a + 5; // a等于a的初始值加5
```

3.1.5 加减 1 的简写

变量加1在C++中非常常见：

```
int x = 0;
x = x + 1;
```

当我们处理像循环那样的操作时，会大量使用这种模式。它的使用非常普遍以至于有一个单独的++操作符，只对变量加1。

上面的代码可以写成：

```
int x = 0;
x++;
```

x的结果是1。++操作符通常称为*递增*操作符，变量加一通常称为变量*递增*。

操作符--的工作原理相同，不过它使变量减1。--操作符通常称为*递减*操作符，变量减1称为变量*递减*。

知道了这一点，你可以猜一下C++的名称是怎么来的？C++基于C语言，字面意思是"C加1"。C++不是一个全新的语言，而是经过补充后的C。我想如果C++的创造者们知道C++其实比C强大那么多，他们可能会把它命名为C平方。

变量赋值使用相似的快捷操作符:

```
x += 5; // x加5
```

同样适用减、乘和除运算：

```
x -= 5; // x减5
x *= 5; // x乘5
x /= 5; // x除以5
```

最后，++和--不但可以用在变量后，还可以用在变量前：

```
--x;
++y;
```

两者的区别是表达式返回的值不同。如果这么写：

```
int x = 0;
cout << x++;
```

输出是0。尽管x修改了，但是表达式x++返回的是x的初始值。因为++在变量的后面，你可以认为变量在被输出后才获取到新值。

如果你把操作符放到变量的前面，就能立即得到新值：

```
int x = 0;
cout << ++x;
```

表达式首先对x加1，接着获取x的值，这样便会输出1。借助这些操作，你可以用C++编写一个小型的计算器了。

```
#include <iostream>

using namespace std;

int main()
{
    int first_argument;
    int second_argument;
    cout << "Enter first argument: ";
    cin >> first_argument;
    cout << "Enter second argument: ";
    cin >> second_argument;
    cout << first_argument << " * " << second_argument << " = " <<
first_argument * second_argument << endl;
    cout << first_argument << " + " << second_argument << " = " <<
first_argument + second_argument << endl;
    cout << first_argument << " / " << second_argument << " = " <<
first_argument / second_argument <<endl;
    cout << first_argument << " - " << second_argument << " = " <<
first_argument - second_argument << endl;
}
```

示例代码6：calculator.cpp

3.2　变量的使用和滥用

3.2.1　C++中声明变量的常见错误

声明变量后可以让程序执行很多操作，但一个错误的变量声明会导致一些初始化错误。例如，如果你想使用一个没有声明的变量，编译会失败，出现变量未声明的编译错误。编译器通常会提示如下的错误：

```
error: 'x' was not declared in this scope
```

如果使用未声明的变量（例子中的x），报错信息取决于你正使用的编译器。示例中的错误信息由MinGW和Code::Blocks产生。

同一个类型可以声明多个变量，但多个变量不能为同一个名称。例如你不能同时用double和int声明my_val。声明两个不同的变量使用同一个名称则会出现类似以下的错误信息：

```
error: conflicting declaration 'double my_val'
error: 'my_val' has a previous declaration as'int my_val'
error: declaration of'double my_val'
error: conflicts with previous declaration 'int my_val'
```

第三个经常出错的地方是行末忘记加分号：

错误代码

```
int x
```

这种错误会导致编译器产生不同的错误信息，错误信息内容取决于变量声明后面的代码。一般来说，编译错误会从变量声明的下一行开始。

最后，还有些错误会发生在运行时，比如你在声明一个变量时变量*未初始化*。那么你必须在使用前进行*初始化*。初始化变量就是在使用前对变量进行赋值。若没有初始化，程序运行结果便会不确定。下面是一个常见的问题程序：


```
int x;
int y;
y = 5;
x = x + y;
```

`y`在使用前被赋值成`5`，但是`x`的初始值却是未知的。程序运行时会随机对`x`进行赋值，因此它可能是任何值！不要想当然的认为变量会被初始化成`0`之类的。

有一个技巧可以避免上述问题，就是在声明变量时直接赋值。

```
int x = 0;
```

这个技巧可以确保变量在创建时便有明确的值。养成这个习惯会让你在以后的编程中减少一些纠结的bug和打字次数。

3.2.2　区分大小写

现在可以讨论另一个容易让你困惑的重要概念了——*区分大小写*。C++区分字符大小写，`Cat`和`cat`对编译器来说是两个不同的东西。在C++中，所有的关键词、函数和变量都区分大小写。

变量在声明和使用时大小写不同（如声明时用X但是使用时用`x`）会导致出现变量未声明的错误，即使你认为已经声明过了。

3.2.3　变量命名

选择有意义、描述性的变量名是非常重要的。下面是一个反面案例：

```
val1 = val2 * val3;
```

这是什么意思？无人可解。等式中的名字几乎没有任何意义。编程当天你会觉得自己写的代码含义很明显，第二天就会感觉完全不可理解了。描述性命名会让你在下次阅读代码时不会糊涂。

例如：

```
area = width * height;
```

就比第一个等式清晰明了，而且结构等式不变，仅仅修改了第一个等式的变量名。

3.3　字符串存储

你可能已经注意到，目前所有的数据类型只允许处理简单的值，比如一个整数或字符。事实上用这些基础数据就可以处理很多的事情，但C++还提供了其他的数据类型。[①]

一个最常用的数据类型是string。string可以存储多个字符。你已经见过将字符串输出到屏幕上了。

```
cout << "HEY, you, I'm alive! Oh, and Hello World!\n";
```

C++ string类允许你对字符串进行保存，修改等操作。

声明字符串也非常容易：

```
#include <string>

using namespace std;

int main ()
{
    string my_string;
}
```

示例代码7：string.cpp

不像你使用其他内置类型，使用字符串时必须使用<string>头文件。因为编译器没有内置string类型，不像整型那样内置在编译器中。字符串类型由C++标准库（一个大型可复用的代码库）提供。

像C++提供的其他基本类型一样，你可以直接使用cin读入用户输入的字符串。

```
#include <iostream>
#include <string>;

using namespace std;

int main ()
{
    string user_name;

    cout << "Please enter your name: ";
    cin >> user_name;
    cout << "Hi "<< user_name << "\n";
}
```

示例代码8：string_name.cpp

程序创建一个字符串变量，提示用户输入他或者她的名字，然后进行输出。

像其他的变量一样，字符串可以进行初始化。

① 事实上C++还可以让你自定义数据类型，不过等我们讲结构的时候再讲这些。

```
string user_name = "<unknown>";
```

如果你想把两个字符串合并，可以用“+”把一个字符串追加到另一个字符串上：

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string user_first_name;
    string user_last_name;

    cout < "please enter your first name: ";
    cin >> user_first_name;
    cout < "please enter your last name: ";
    cin >> user_last_name;
    string user_full_name = user_first_name + " " + user_last_name;

    cout < "your name is: " << user_full_name << "\n";
}
```

示例代码9：string_append.cpp

这段程序把用户的姓、空格、用户的名这三个单独的字符串合并成一个字符串。[1]

如果你想一次读取一整行字符串，可以使用一个特殊的函数getline，它用来读取整行数据。这个函数甚至可以帮你自动丢弃末尾的换行符。

使用getline，你需要传入输入源（本例中是cin）、读入字符串和终止字符三个参数。例如，下面代码可以读取用户的名。

```
getline( cin, user_first_name, '\n' );
```

getline也可以用来读取某个字符之前的输入。比如逗号之前（尽管程序还是要用户按回车键之后才能接受数据）：

```
getline( cin, my_string, ',' );
```

如果用户输入：

```
Hello, World
```

my_string会赋值为“Hello”，本例中剩下的文本“World”将会驻留在输入缓存中，直到另一个输入声明读取它。

① 术语提示：有时候你会看到用单词**concatenate**表示两个字符串连接到一起。concatenate来自拉丁语的“to chain together”，catena 在拉丁语中表示“链接”。

3.4 基本类型的存储解析

注意：这部分内容是高级知识，你目前还不需要使用。如果对这部分内容比较迷糊，请先跳过，回头再看。

此时，您可能想知道为什么我们会有如此多不同类型的基本变量。

让我们花点时间学习一下所有计算程序的两个基本构建块：位（bit）和字节（byte）。位是计算机存储的基本单元，一个位就是一个开关，根据开关的设定，表示1或0。1字节由8个位构成，相当于有8个位置，每个位置都可以代表两个值，那么一共就有256种0和1的组合方式。让我们分解一下。一个位可以存储0或1两个值，两个位能存储一个位的两倍：00、01、10和11。三个位是两个位的两倍，在两位的组合上又添加了一个0或1。所以每多一个位就能让代表的值数量翻倍。换而言之，对于n个位，我们可以表示2^n个值。1字节是8个位，所以它有2^8种组合。如果有2字节，那么就是16个位，可以代表2^{16}（65536）个值。

看不懂上面的内容不要紧，主要思想是字节越多，能存储的东西就越多。

例如，`char`是单字节，一个`char`只能存储256种不同的数据。而一个整型通常占用4字节，也就是说它能够表示大约40亿的数据。

有一个很好的例子，`double`和`float`的不同点仅仅在于`double`占用的空间是`float`的两倍。`float`是存储小数的原始变量类型，`float`的命名事实上也是来源于小数点可以“浮”在数字的不同位置。换言之，你可以有4个小数2个整数（12.234 5），或者4个整数2个小数（3 421.12）。小数点前和小数点后的数字都没有限制。

如果你一下无法接受这些，不要紧，它们都是历史了。只要知道浮点数就代表着“带有小数点的数”。`float`只有4字节，而`double`有8字节，所以`float`比`double`存储的少。以前的电脑内存很少，4字节是一个很大的数，程序员会竭尽全力节省空间。但现在，多使用`double`会更好一些。不过当程序可用内存较小时（如手机中的小内存），你仍然需要选择使用`float`。

`char`是最小的数据类型，它只有1字节。你可能会想，既然空间大小无所谓，为什么还需要char呢。因为`char`有特殊的意义——输入输出都用字符而不是数字。用户可以向`char`变量输入字符，而且在输出字符时，你会更希望直接显示存储在变量里的数字代表的字符，而不是显示这些数字。你可能会疑惑“这是什么意思？为什么字符会是数字呢”，原因在于计算机用数字的形式存储我们看到的字符（如字母“a”）。有一个数字和字符之间的映射表，称为ASCII表。ASCII表用来查找每个数字代表什么字符。当程序要输出字符而不是数字时，程序会先从ASCII表中查询该数字对应的字符①。

① 我不得不提示一下ASCII表非常的小，它只有256个值。也就是说它不适合像日语或汉语这种超过256个字符的语言。处理这些语言采用Unicode编码。这超出了本书的范围。你可以在下面的网址了解相关信息：http://www.cprogramming.com/tutorial/unicode.html。

1. 浮点数的缺陷

我想让你了解一些关于浮点数的东西。能使用float或double听起来是很不错，因为它们能表示的值范围很大。比如double能表达的最大数约是1.8×10^{308}。这是一个有308个0的数字。但一个double变量只有8字节，它只能存储2^{32}（18 446 744 073 709 551 616）种可能值（这个数字也很大，但远没有308个0）。

事实上，一个double只能表达1 800亿亿个数字，这个数字特别大，以至于我需要查找什么数量级可以表示有18个0的数。但它依然不是308个0。浮点数用一种类似科学计算法的格式计算出一个范围，它只能表达范围内的数。

在科学计数法中，你用$x \times 10^y$来表示数字。x通常存储数的前几位，而指数y，则用来提高数的数量级。例如，地球和太阳之间的距离可以写成$9.295\,6 \times 10^7$英里（大约9300万英里）。

指数越大，电脑能存储的数也就越大。但是非指数部分没办法存储300个数字，它只能存储15个，所以只能使用15位精度的浮点数。当处理比较小的数时，真实数值和电脑存储的数误差非常小。而当处理大数时，虽然相对误差小，但绝对误差会非常大。例如，如果精度只有两位，你可以写成地球距离太阳9.3×10^7英里，相对而言，它非常接近正确值（不到0.1%的误差）。但如果换算成绝对距离，相差了4.4万英里，这接近地球周长的两倍。当然，这只用了两位精度，如果用15位精度，能比较精确地表达百万级数字。

大多数情况下，浮点数不精确不会影响你。但如果正在处理严谨的数值运算或科学计算，这便会关系重大。

2. 整数的缺陷

整数也有缺陷。事实上，整数和浮点数一点都不兼容。不像浮点数，整数会准确存储你输入的值。但它不会接受小数点。当浮点数和整数一起运算时，结果会是整数。它会被截断，非小数部分保留，剩下的丢弃。

举个例子，如果你在数学考试中回答$5/2 = 2$，那肯定考试不及格。但计算机确是一直这么运算的！你需要使用非整型的数据类型来获得带有小数点的答案。

程序默认输入的数字为整型，这也就是为什么5/2会被计算成2。不过如果数字中包含小数点，比如5.0/2.0，编译器就会按照浮点数进行计算，然后返回你期望的结果：2.5。

3.5　问答题

(1) 什么类型可以存储数值3.131 5?

A. int　　B. char　　C. double　　D. string

(2) 下面哪个是比较两个变量的操作符?

A. :=　　B. `=`　　C. `equal`　　D. `==`

(3) 如何获取`string`数据类型?

A. 语言中包含，无需任何操作
B. 因为字符串用在输入输出上, 你需要引用`iostream`头文件
C. 引用`string`头文件
D. C++不支持

(4) 下面哪个变量类型不正确?

A. `double`　　B. `real`　　C. `int`　　D. `char`

(5) 怎么读取用户的一整行输入?

A. 使用 `cin >>`　　B.使用 `readline`　　C. 使用 `getline`　　D. 很困难

(6) C++中，`cout << 1234/2000` 会输出什么结果?

A. 0
B. 0.617
C. 大约 0.617, 不过结果不能精确的存储在浮点数中
D. 要看等式两边的类型

(7) 为什么C++在有整数类型的情况下还需要`char`类型?

A. 因为字符和整数是两种完全不同的类型，一个是数字，一个是字母
B. 为了向下兼容C
C. 字符比数字更加容易读入和输出，尽管字符实际上存储为数字
D. 对国际支持，处理像汉语和日语这种包含很多字符的语言

3.6　实践题

(1) 编写程序输出你的名字。

(2) 编写程序读取两个数字并相加。

(3) 编写程序，读取用户输入的两个数字进行相除，获取准确的结果。确保整数和小数都能正确计算。

第 4 章

if语句

目前你已经学会了编写按顺序执行的程序，程序还无法根据用户的不同输入采取不同的操作。if语句可以控制程序根据给定条件的是（true）或非（false），来判断是否执行某段代码。换而言之，if语句允许程序根据用户的输入选择不同的操作。例如，程序可以通过if语句判断用户输入的密码是否正确，从而决定用户能否访问程序。

4.1 if 的基础语法

if语句的结构非常简单：

```
if ( <表达式的值为true> )
      执行这个语句
```

或：

```
if ( <表达式的值为true> )
{
      执行花括号内的所有语句
}
```

紧跟着if语句（将被选择性执行）的代码称为if语句的函数体（就像main函数里的代码称为main函数的函数体）。

下面是if语法的一个简单示例：

```
if ( 5 < 10 )
    cout << "Five is now less than ten, that's a big surpise";
```

这里，我们需要判断语句“5小于10”是否正确。当然结果肯定是正确的。你可以引用iostream头文件，编写一个完整的程序，把上面的代码放到main函数中运行一下，看看结果如何。

下面是一个用花括号包含多行语句的示例程序：

```
if ( 5 < 10 )
{
```

```
        cout << "Five is now less than ten, that's a big surprise\n";
        cout << "I hope this computer is working correctly.\n";
    }
```

如果if语句后有多行代码，请用花括号将它们括起来，确保当且仅当if语句判断为真时，花括号里的所有代码都能执行。我推荐你在编写if语句函数体时使用花括号。这么做可以确保所有应当执行的语句都被包含。同时也会让if语句的函数体更清晰易读。在if函数体中不使用花括号包含第二个if语句是常见的错误，这会导致第二个if语句一直被执行。

```
if ( 5 < 10 )
    cout << "Five is now less than ten, that's a big surprise\n";
    cout << "I hope this computer is working correctly.\n";
```

代码缩进让人很难发现这类错误。相比之下，习惯把语句放在花括号内会安全很多。

目前为止，我描述的if语句语法比较枯燥，接下来让我们看看处理用户输入的实际的if语句。

```
#include <iostream>

using namespace std;

int main ()
{
    int x;
    cout << "Enter a number: ";
    cin >> x;
    if ( x < 10 )
    {
        cout << "You entered a value less than 10" << '\n';
    }
}
```

示例代码10：variable.cpp

这个程序和之前的示例程序不同，它比较的值来自用户输入，而不是像之前的程序把值固定在程序里。这很令人兴奋！程序第一次可以根据用户输入执行完全不同的操作。现在，让我们看看if语句的灵活性。

4.2 表达式

if语句是一个简单的表达式。表达式是单个或多个相联的计算单个值的语句。大部分能读取变量或常量（如数字）的语句都能读取表达式。事实上，变量和常量也是表达式——简单的表达式。加法操作、乘法操作是稍微复杂一点的表达式。当用在比较上时，表达式会返回true或者false。

4.2.1 truth

对于诗人来说，真就是美，美就是真理，在文学里你也只需要知道这些[①]但编译器不是诗人。对编译器来说，表达式返回非零的数便是true，返回零就是false。比如语句：

```
if ( 1 )
```

能让if语句函数体里的所有代码都被执行。但是语句：

```
if ( 0 )
```

便会使函数体里的所有代码都不被执行。

C++有两个特殊的关键字——true和false，你可以将它们直接写在代码中。当按整型输出时true输出1，false输出0。

当你用关系操作符执行比较时，操作符也将返回true或false。例如0 == 2的计算结果为false，2 == 2的计算结果是true。（注意，判断相等时使用两个等号==，使用一个等号是对变量赋值。）将关系操作符用在if语句中时，关系表达式的结果可以直接对应到true或者false无需再进行检查：

```
if ( x == 2 )
```

等同于：

```
if ( ( x == 2 ) == true )
```

第一种更易读。

编程时，我们时常需要比较两个变量值之间的大小关系。

下面这个表列出了用于两个值之间比较的关系操作符。

>	大于	5>4是true
<>	小于	4<5是true
>=	大于等于	4>=4是true
<>	小于等于	3<=4是true
==	等于	5==5是true
!=	不等于	5!=4是true

① 参见http://www.bartleby.com/101/625.html。

4.2.2　布尔型

C++用一个特殊的类型bool存储比较的结果[1]。bool类型和整型没有什么不同，但它非常的清晰明了，因为它只有两种值——true和false，这是它的优点。这些关键字和bool变量能让你的思路更清晰。注意，所有比较操作的返回值都是布尔值。

```
int x;
cin >> x;
bool is_x_two = x == 2; // 注意，双等号表示比较

if ( is_x_two )
{
    //因为x等于2，所以程序会执行到这里！
}
```

4.3　**else** 语句

很多时候，你想让程序在执行操作前先进行一个简单的判断，如果判断为true（比如用户输入的密码是正确的），执行一种操作，如果判断为false（比如用户输入的密码是错误的），执行另一种操作。

else语句允许你执行if-else比较。如果if语句里的条件为false，else之后代码便会执行（可能是一行，也可能是花括号内的多行）。下面的示例程序将判断用户输入的数是负数还是正数。

```
#include <iostream>

using namespace std;

int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;
    if ( num < 0 )
    {
        cout << "You entered a negative number\n";
    }
    else
    {
        cout << "You entered a non-negative number\n";
    }
}
```

示例代码11：non_negative.cpp

① bool以George Boole命名。布尔逻辑是设计数字计算机的基础，用true和false两种值表示的一种逻辑运算。George Boole是设计布尔逻辑的数学家。

4.4 else-if

else的另一类用法是当有多个条件语句同时为true时，你只想执行其中某一个条件语句。例如，你可能想让上面的示例代码检测三种不同的情况：负数、零和正数。你可以在if语句和它的函数体后使用else-if语句。在这种方式下，如果第一个语句为true，后面的else-if将会被忽略，如果if语句为false，程序便会判断else-if语句的条件，如果该条件为true，后面的else语句也不会执行。编程中可以使用一系列else-if语句确保只有一个代码块执行。

下面让我们修改上面的代码，使用一个else-if来判断零值：

```
#include <iostream>

using namespace std;

int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;
    if ( num < 0 )
    {
        cout << "You entered a negative number\n";
    }
    else if ( num == 0 )
    {
        cout << "You entered zero\n";
    }
    else
    {
        cout << "You entered a positive number\n";
    }
}
```

示例代码12：else_if.cpp

4.5 字符串比较

C++中的string类允许你使用之前几章学习的所有用于比较的方法。利用string类的比较，我们可以编写下面的代码检查程序。

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string password;
```

```
    cout << "Enter your password: " << "\n";
    getline( cin, password, '\n' );
    if ( password == "xyzzy" )
    {
        cout << "Access allowed" << "\n";
    }
    else
    {
        cout << "Bad password. Denied access!" << "\n";
        //使用return可以方便地结束程序
        return 0;
    }
    // 继续执行!
}
```

示例程序13：password.cpp

程序读取用户的输入，和密码xyzzy进行对比。如果输入的内容和密码不同，程序便会立即从main函数返回。[①]

你也可以使用其他的比较操作，比如按照字母序比较两个字符串大小，或使用 != 判断两个字符串不相同。

4.6　逻辑运算符在条件语句上的有趣应用

目前，我们一次只能判断一个条件。如果想同时执行两次判断，比如判断用户名和密码都正确，你就不得不写很多if-else语句。幸运的是，C++包含*逻辑运算符*，它提供了同时执行多个判断的功能（名字和之前的bool型有关，逻辑运算符作用于布尔值）。

你可以使用逻辑运算符编写更复杂的判断语句。例如，如果想判断一个名为age的变量值是否大于5且小于10，你可以使用逻辑与（Boolean AND）确保age>5和age<10都为true。

逻辑操作符和比较操作符一样，根据表达式的结果返回true或者false。

4.6.1　逻辑非

逻辑非（Boolean NOT）只有一个输入，如果输入为true，那么返回false，输入为false，则返回true。例如，是（true）计算结果为false，非（false）计算结果为true。零之外的任何数字的非值都为false。

C++中非的符号是!（没错，就是感叹号）。

例如：

① 当然，真正的密码检查程序不会这么简单，首先，你不会把密码直接放进源代码里。

```
if ( ! 0 )
{
    cout << "! 0 evaluates to true";
}
```

4.6.2 逻辑与

如果两个输入值都为true，那么逻辑与返回true（即“第一个值”与“第二个值”都为true）。true与false结果为false，因为其中一个输入值为false（两个值都是true结果才是true）。任意非零数字与false进行逻辑与返回值为false。

C++中与的操作符是&&，不要认为它用来判断两个数是否相等。它只用来判断两个参数是否都为true。

```
if ( 1 && 2 )
{
    cout << "Both 1 and 2 evaluate to true";
}
```

短路求值

如果第一个表达式是布尔型且返回false，那么第二个表达式将不会被计算。这就是短路求值。

短路运算很有用，你可以写出当且仅当第一个条件为true时才判断第二个条件的表达式。例如下面的if语句中，使用短路预算可以在判断10除以x小于2时避免除以零。

```
if ( x != 0 && 10 / x < 2 )
{
    cout << "10 / x is less than 2";
}
```

当运行到if语句时，程序首先会判断x是不是0，如果是0，便直接跳过，不会判断第二个条件。也就是说，你不需要担心除零引起程序崩溃。如果没有短路运算，不得不这样写：

```
if ( x != 0 )
{
    if ( 10 / x < 2 )
    {
        cout << "10 / x is less than 2";
    }
}
```

使用短路运算，你可以写出清晰明了的代码。

4.6.3 逻辑或

如果两个值都为true或其中一个为true，逻辑或（Boolean OR）返回true。例如，true

或false返回true。false或false返回false。C++中逻辑或写成||，是管道符。在键盘上，它们被标记为中间有间隔的竖条，尽管大部分字体把它们显示成没有间隔的竖条。大部分键盘上管道符和\符号在一个键上，需要按下Shift键才能输出。

和逻辑与一样，逻辑或也可以进行短路计算，如果第一个条件为true，便不会检查第二个。

4.6.4　综合表达式

利用基本的逻辑运算符，你一次能判断两个条件。如果想判断更多呢？还记得表达式是由变量、操作符和常量构成的吗？表达式同样也能由其他表达式构成。

例如，你可以用逻辑与和双等号比较操作符判断x等于2且y等于3。

```
x == 2 && y == 3
```

分析一段使用布尔值同时检查用户名和密码的示例程序：

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string username;
    string password;
    cout << "Enter your username: " << "\n";
    getline( cin, username, '\n' );

    cout << "Enter your password: " << "\n";
    getline( cin, password, '\n' );
    if ( username == "root" && password == "xyzzy" )
    {
        cout << "Access allowed" << "\n";
    }
    else
    {
        cout << "Bad username or password. Denied access!" << "\n";
        //return是终止程序的有效方法
        return 0;
    }
    //继续执行
}
```

示例程序14：username_password.cpp

程序运行时只允许输入正确密码的名为root的用户访问。你可以用else-if语句拓展程序，使其允许多个不同用户访问，每个用户拥有自己的密码。

优先级

之前的例子中包含几个子表达式：

```
username == "root"
```

和：

```
password == "xyzzy"
```

C++中，操作符需要按照优先级进行计算。算术操作符的优先级（+、-、/和*）和普通的数学运算一样：乘法和除法的优先级大于加法和减法。

对于逻辑操作符，非操作优先，紧接着是比较操作，逻辑与比逻辑或优先。

下表中列出了逻辑操作符和比较操作符的优先级顺序

!
==, <, >, <=, =>, !=
&&
||

你可以用括号控制逻辑操作符和算数运算符的运算顺序。

例如，我们之前的例子：

```
x == 2 && y == 3
```

如果想实现“条件非true”，可以用括号：

```
! ( x == 2 && y == 3 )
```

4.6.5 逻辑表达式示例

让我们分析一些更复杂的逻辑表达式，看看你是否已经掌握了逻辑运算符。

下面表达式结果是什么呢?

```
! ( true && false )
```

结果是true。因为true && false结果为false，而!false结果为true。

下面还有一些题目，答案在脚注：

```
! ( true || false )①
```

① false。

```
! ( true || true && false )
! ( ( true || false ) && false )
```
[①] [②]

4.7 问答题

(1) 下面哪个是`true`?

A. `1`　B. `66`　C. `.1`　D. `-1`
E. 以上全部

(2) 下面哪个是逻辑与的操作符?

A. `&`　B. `&&`　C. `|`　D. `|&`

(3) 表达式`!( true && ! ( false || true ) )`的结果是?

A. `true`　B. `false`

(4) 下面哪个是`if`语句的正确语法?

A. `if expression`　B. `if { expression`
C. `if ( expression )`　D. `expression if`

4.8 实践题

(1) 编写程序，要求用户输入两个用户的年龄，并指出谁的年龄更大；需要处理超过100的输入。

(2) 编写一个简单的数字密码系统，两个数都有效时解密。要求只使用一个`if`语句进行判断。

(3) 编写一个小型计算器，输入4个算术运算符中任一个和进行运算的两个参数，输出计算结果。

(4) 拓展本章中的密码检验程序，使其可以处理多用户，每个用户有自己的密码，确保用户名和密码一一对应。用户第一次登录失败时提示重新登录。思考处理多个用户和密码的方法难度。

(5) 思考哪种语言结构或语言特性可以让添加新用户更简单，且不需要重新编译程序。（注意：你不需要用目前学到的C++知识解决这个问题，问题的目的是让你思考怎么使用接下来几章中学习到的工具。）

---

① `false`（非之前的结果是`true`）。

② `true`。

第 5 章 循环

目前为止，你已经学会了如何让程序根据用户的输入执行不同的操作，但程序仍然只能运行一次。你还无法编写程序反复提示用户重新输入。上一章后面有一道密码程序实践题，要求你在用户密码输入错误后提示重新输入，对于这道题，不得不编写一连串if语句来重新核对密码，无法在用户输入正确密码前允许用户重新输入密码。

这就是循序要解决的事情。循环可以重复执行某个代码块，功能极其强大，是大部分程序的核心。大部分程序或网站产生的极其复杂的输出（如留言板）本质上是多次执行一个简单的任务。现在，让我们想一下这意味着什么：循环可以让你编写的简单语句重复执行从而产生大量结果。你可以按照用户意愿反复提示他重新输入密码，也可以在互联网论坛上显示上千份帖子。这非常的赞！

C++有三种循环：while、for和do-while。每种用法略有不同，我们一个一个地学习。

5.1 **while** 循环

while循环是最简单的一种循环，基本结构是：

```
while ( <条件> ) { 当条件为true时执行的代码 }
```

事实上，除了会让自身重复执行外，while循环和if语句非常像，控制条件也是布尔表达式。例如，下面是一个包含两个控制条件的while循环：

```
while ( i == 2 || i == 3 )
```

下面是一个基本的while循环示例：

```
while ( true )
{
    cout << "I am looping\n";
}
```

警告：如果运行这个循环，它会永不停止！因为条件一直为true。这称为**无限循环**，无限循环永不停止，只有杀掉程序才能终止运行（可以通过按下Ctrl-C、Ctrl-Break或关掉控制台窗口杀死程序完成）。为了避免无限循环，你要确保循环条件不会一直为true。

常见错误

一个导致无限循环的常见错误是将循环控制条件中的双等号误写成单等号。

错误代码

```
int i = 1;
while ( i = 1 )
{
    cin >> i;
}
```

这个循环试图读取除1之外的用户输入，遗憾的是，循环条件是：

```
i = 1
```

而不是：

```
i == 1
```

表达式i = 1只会把i赋值为1。而赋值表达式只会返回分配给它的值，此例中表达式返回1。因为1不是零，表达式为true，所以这个循环将会无限执行下去。

让我们看看功能正常的循环！下面是一段完整的循环示例程序，程序输出从0到9的数字：

```
#include <iostream>

using namespace std;

int main ()
{
    int i = 0; // 不要忘记声明变量

    while ( i < 10 ) // 如果i小于10就循环
    {
        cout << i << '\n';
        i++;             // 因为满足条件，所以i值增加
    }
}
```

示例代码15：while.cpp

如果你对循环依然困惑，可以试着这么想：当程序运行到循环体最后的括号时，会跳转到循环的开头，重新判断条件，根据真假决定是再次重复，还是停止循环跳转到下一条语句。

5.2 for 循环

for循环非常灵活方便，其语法是：

```
for ( 变量初始化; 条件; 变量更新 )
{
    当条件为true时执行此处代码
}
```

循环内可以有很多内容，让我们分析一个短小的示例，分解循环中的每个元素。事实上，for循环和上面的while循环较为相似：

```
for ( int i = 0; i < 10; i++ )
{
    cout << i << '\n';
}
```


5.2.1 变量初始化

此例中，变量初始化是int i = 0，变量初始化允许编程人员声明一个变量并且赋值（或者对已经存在的变量进行赋值）。这里，我们声明了变量i。当某个变量的值在循环中被反复判断时，这个变量称为循环变量，此例中的i便是循环变量。编程中经常使用字母i和j作为循环变量。每经过一次循环值增加1的变量称为循环计数器，变量从一个值计数到另一个值。

5.2.2 循环条件

当变量表达式为true时，循环条件控制程序重复自身（就像while循环一样）。此例中，我们计算x是否小于10。和while循环一样，程序在执行循环前会判断条件，每次循环结束也会重新判断，决定是否继续循环。

5.2.3 变量更新

在变量更新部分循环变量将被更新。用于变量更新的可能是表达式，如i++,i=i+10，也可能是函数调用，比如你可以调用一个不改变变量名但对代码有效的函数。

因为多数循环只有一个变量、一个条件和一个变量更新。for循环把所有和循环相关的逻辑写在一行中，这种方式非常紧凑。

注意这紧凑的一行使用分号分割各个部分；你不能忘掉分号。任何单个甚至所有部分都可以为空，但是分号必须存在。如果条件为空，条件默认为true，程序将会一直循环，直到其他操作将其终止。这是另一种编写无限循环的方法。

想真正了解for循环的每个部分，我们可以和之前的while循环进行对比，让它们做相同的操作：

```
int i; // 变量声明和初始化
while ( i < 10 ) // 条件
{
    cout << i << '\n';
    i++; // 变量更新
}
```

for循环更加紧凑。

让我们来看另一个for循环的示例，它能做一些比单纯输出数字更有趣的事情。下面是完整的程序，将输出从0~9的平方数：

```
#include <iostream>

using namespace std;

int main ()
{
    //当i<10循环执行，每次循环后，i加1
    for ( int i = 0; i < 10; i++ )
    {
        //请记住在下次循环前，程序会判断条件语句
        //因此，当i等于10时循环结束。
        //i在判断条件之前被更新
        cout << i << " squared is " << i * i << endl;
    }
}
```

示例代码16：for.cpp

这段程序是for循环的一个非常简单的示例。让我们分析示例，学习for循环的每个部分是如何执行的。

(1) 执行初始化步骤：i被设为0。
(2) 判断控制条件：因为i小于10，执行循环体。
(3) 变量更新：i加1。
(4) 判断条件，若条件非true，循环结束。
(5) 若条件为true，执行循环体，重复所有的内容。跳回第(3)步直到i不再小于10。

记住变量更新的步骤只发生在循环体运行之后，并不是第一时间就执行。

5.3　**do-while** 循环

do-while循环比较具有目的性且很少使用。do-while循环的主要目的就是编写至少要执行

一次的循环体。结构为：

```
do
{
    //循环体……
} while (条件) ;
```

循环条件在函数体的末尾而不是开头；因此，循环体至少要执行一次，如果执行一次之后循环条件依旧为true，程序将跳转到循环体的开头再次执行。do-while循环基本上算是颠倒的while循环。while循环是“当条件为true，执行循环体”，do-while循环是“执行循环体，如果条件为true跳转到开头重新执行”。下面是一个简单的示例，让用户反复输入密码直至密码正确。

5

```
#include <string>
#include <iostream>

using namespace std;

int main ()
{
    string password;
    do
    {
        cout << "Please enter your password: ";
        cin >> password;
    } while ( password != "foobar" );
    cout << "Welcome, you got the password right";
}
```

示例代码17：dowhile.cpp

这个循环将执行循环体至少一次，允许用户输入密码；如果密码不正确，循环将会继续，提示用户再次输入密码，直到用户输入正确的密码。

注意上面例子中while之后的分号！编程时很容易忘记分号，因为其他的循环不需要分号；事实上，其他的循环不能以分号结尾以免增添混乱。

5.4　控制循环

大多数情况通过判断循环条件退出循环，但有时候你也想早点跳出循环。C++正好有一个关键字：**break**。break语句可以立即终止循环，无论循环执行到哪一步。

下面这段代码使用break语句重写密码示例程序，结束它的无限循环：

```
#include <string>
#include <iostream>

using namespace std;
```

```
int main ()
{
    string password;
    while ( 1 )
    {
        cout << "Please enter your password: ";
        cin >> password;
        if ( password == "foobar" )
        {
            break;
        }
    }
    cout << "Welcome, you got the password right";
}
```

示例代码18：break.cpp

break语句会立刻结束循环，跳转到结束括号。此例中一旦输入正确密码循环便会终止。因为break语句可以出现在循环的任何地方，包括最后，你可以像我这样编写无限循环代替do-while循环。break语句和do-while循环末尾的条件判断语句功能非常相似。

当需要从庞大的循环体中跳出时，break语句非常实用。但是太多的break语句会让代码难以阅读。

第二种控制循环的方法是使用continue跳出单次循环。当运行到continue语句时，当前的单次循环提前结束，但循环并没有退出。例如，你可以借助continue语句编写一个不输出数字10的循环。

```
int i = 0;
while ( true )
{
    i++;
    if ( i == 10 )
    {
        continue;
    }
    cout << i << "\n";
}
```

上面的代码中，循环永不停止，但是当i增加到10时，continue语句会使程序跳过cout调用，跳回循环的开始行，重新判断循环条件。在for循环中使用continue时，continue语句之后会立刻进行变量更新。

当你想跳过循环体中间的某些代码时，使用continue语句非常有用。例如，当判断用户输入时，如果用户输入了错误信息，可以使用下面的循环结构跳过输入处理：

```
while ( true )
{
    cin >> input;
```

```
    if ( ! isValid ( input ) )
    {
        continue;
    }
    //像平常一样处理输入
}
```

5.5 嵌套循环

在C++中，很多时候不止循环一个值，可能同时循环两个不同但相关的值。例如，你可能想在一个循环中输出一串论坛的帖子，每个帖子都包含很多不同的值，如帖子的主题、作者和正文。你可以用第二个循环输出这些信息，但是第二个循环需要嵌套到其他的循环中。这种循环就称为*嵌套循环*，表示一个循环嵌入在另一个里面。

让我们看一个不像论坛帖子那么复杂的简单例子：使用嵌套循环输出一个乘法表：

```
#include <iostream>

using namespace std;

int main ()
{
    for ( int i = 0; i < 10; i++ )
    {
        cout << '\t' << i; // \t代表tab符，可以对输出的数据格式化

    }

    cout << '\n';

    for ( int i = 0; i < 10; ++i )
    {
        cout << i;

        for ( int j = 0; j < 10; ++j )
        {
            cout << '\t' << i * j;
        }
        cout << '\n';
    }
}
```

示例代码19：nested_loops.cpp

当你使用嵌套循环时，可以用*外循环*和*内循环*区分两个循环。此例中，包含变量j的是内循环，包含变量i的是外循环。

请注意内循环和外循环中不能使用相同的变量：

```
错误代码
for ( int i = 0; i < 10; i++ )
{
    //哎呀，i不小心重复定义了
    for ( int i = 0; i < 10; i++ )
    {
    }
}
```

你可以嵌套两个以上的循环，外循环嵌套内循环，内循环再嵌套一个循环，一层层下去，随便嵌套多少层。

5.6　选择合适的循环

目前你已经学了C++的三种不同的循环，但可能会奇怪：为什么需要三种循环呢？

事实上，你并不真的需要三种循环。像do-while循环大部分出现在课本上，在实际编程中for循环和while循环更为普遍。

下面的内容是关于选择合适循环类型的快速指南。注意，它们只是一些经验总结，随着时间的推移，对依据代码选择合适类型的循环会有更深的了解，不要让这个指南成为你的金规玉律。

5.6.1　**for**循环

当你知道循环的准确次数时可以使用for循环，例如从0计数到100，用for循环计算乘法表也非常完美。for循环更是遍历数组的标准方式（关于数组，参见第10章）。相反，当变量的更新运算比较复杂时不建议使用for循环，for循环适用于语句单一准确的情况，如果变量更新的步骤需要多行代码，使用for循环就会失去优势。

5.6.2　**while**循环

取长补短！如果循环条件比较复杂，或者在获取循环变量下一个值前需要做很多的数学运算，可以考虑while循环。while循环可以清晰的看到循环什么时候结束，但是很难看出每次循环后哪里发生了变化。如果变化比较复杂，最好使用一个while循环，至少读者会知道这不是一个简单的更新。

例如，如果你有两个不同的循环变量：

```
int j = 5;
for ( int i = 0; i < 10 && j > 0; i++ )
{
    cout << i * j;
```

```
    j = i - j;
}
```

注意，不是所有影响循环的代码都会放在for循环的单行中，有一些会被放在循环体的末尾。这可能会误导读者，因此最好选择while循环进行处理。

```
int i = 0;
int j = 5;

while ( i < 10 && j > 0 )
{
    cout << i * j;
    j = i - j;
    i++;
}
```

这依然不完美，但至少不会误导读者。

编写接近无限循环的程序时也适合使用while循环。例如，你有一个国际象棋程序，希望对战双方在游戏结束时都能成为赢家。

5.6.3 **do-while**循环

do-while是编程的黑天鹅[1]——它们很长时间才出现一次。使用do-while循环的唯一原因是你想执行至少一次操作。前面的提示用户输入密码的示例程序是一个很好的应用场景，或者更普遍的，任何需要用户输入且重复提示直至用户输入正确密码的用户交互程序都适合使用do-while。在某些情况中，如果想让循环体重复，但后面运行时需要和第一次运行不同，它也可能不是一个最好的选择——例如在用户输入错误密码时你想提示不同的信息。

例如，下面的代码如何用do-while循环实现？

```
string password;

cout << "Enter your password: ";

cin >> password;
while ( password != "xyzzy" )
{
    cout << "Wrong password--try again: ";
    cin >> password;
}

string password;

do
{
```

① 黑天鹅在这里指的是难以预见的事情。——译者注

```
    if ( password == "" )
    {
        cout << "Enter your password: ";
    }
    else
    {
        cout << "Wrong password--try again: ";
    }
    cin >> password;
} while ( password != "xyzzy" );
```

想想do-while循环是如何使代码更复杂的？关键点在于循环体不一样，尽管都在读取用户的输入，但我们想对用户显示不同的信息。

5.7 问答题

(1) 代码`int x; for(x=0; x<10; x++) {}`中，x最终的值是？

A. 10　　B. 9　　C. 0　　D. 1

(2) `while(x<100)`之后的代码何时会执行？

A. 当x小于100　　B. 当x大于100
C. 当x等于100　　D. 当它愿意的时候

(3) 哪个不是循环结构？

A. for　　B. do-while　　C. while　　D. repeat until

(4) do-while能保证循环几次？

A. 0　　B. 无限次　　C. 1　　D. 不定

5.8 实践题

(1) 编写程序输出完整的“99 Bottles of Beer”的歌词[1]。
(2) 编写一个菜单程序，允许用户从列表中选择，如果输入不在列表选项内，重新输出列表。
(3) 编写程序计算用户输入的所有数的和，当用户输入0时结束程序。
(4) 编写密码提示，只允许用户尝试特定的次数——让用户无法轻易编写密码破解程序。
(5) 尝试用不同的循环编写每道实践题——观察每种循环适用于哪类问题。
(6) 编写程序输出前20个数的平方数。

---

① 如果你不知道这首歌，歌词在这里：http://en.wikipedia.org/wiki/99_Bottles_of_Beer。

(7) 编写一个调查程序，统计三种可能结果的出现次数。第一个输入是调查的问题；接下来的三个输入是可能的结果。第一种结果用1表示，第二种用2，第三种用3。统计所有的结果直到输入0。当输入结束后程序会显示调查的结果。请尝试用条形图输出结果，确保无论输入多少个结果，条形图都能适应屏幕输出。

第 6 章 函　数

6

在上一章中我们学习了循环，现在已经可以编写一些有趣的程序。不过所有的代码都必须写在main函数里。如果你想在main函数里编写复杂的程序，那么程序必将庞大且晦涩难懂。在完成前面几章的某些复杂练习题时，你应该已经注意到了这个问题。再者，如果想在程序的不同位置做相同的事情，你就不得不一遍遍的复制粘贴这些代码。

函数此时要登场了——通过把程序分解成函数，你可以在很多地方不复制粘贴就能复用这些代码。事实上，在前面的几章中你已经使用过几个标准函数，用它们处理输入和输出。

目前所学已经足够你编写一个新程序。函数的功能是组织代码，它能使代码方便复用且容易阅读。

6.1 函数语法

通过前面的学习，你已经知道如何创建一个函数；每个程序至少要有一个main函数！

接下来我们将学习另一类函数，详解这类函数中的每个部分。

```
int add (int x, int y)
{
    return x + y;
}
```

OK，开始！首先，请注意到上面的函数和我们熟悉（之前写过多次）的main函数非常像。它们只有两点不同。

(1) 上面的函数有两个参数：x和y。main函数没有任何参数。
(2) 这个函数显式地返回一个值。（记住，main函数也有一个返回值，但在程序中不需要使用return语句。）

代码行：

```
int add (int x, int y)
```

首先给出了返回值类型，然后给出了函数名，函数名后紧接着的是括号内的两个参数。如果函数没有参数，你可以直接写一对空括号，如下：

```
int no_arg_function ()
```

如果函数没有返回值，那么返回值类型声明为void，如直接向屏幕输出信息的函数。void可以防止你把函数用作表达式（表达式用在变量赋值或if语句判断条件中）。

返回值由return语句提供；这个函数中它只有一行：

```
return x + y;
```

但你可以像main函数一样编写多行，函数运行到return语句就会停止，把值返回给调用它的代码。

一旦声明了函数，就可以像下面这样调用它：

```
add( 1, 2 ); // 忽略返回值
```

你也可以把函数用作表达式，对变量赋值或者直接输出：

```
#include <iostream>

using namespace std;

int add (int x, int y)
{
    return x + y;
}
int main ()
{
    int result = add( 1, 2 ); //调用add函数，将返回值传递给变量result
    cout << "The result is: " << result << '\n';
    cout << "Adding 3 and 4 gives us: " << add( 3, 4 );
}
```

示例代码20：add_function.cpp

在这个例子中，程序看起来像是的cout输出add函数的值，与输出变量不同， cout在这个例子中输出的是表达式的计算结果，而不是字符串add(3, 4)。程序执行结果和下面这行代码一样。

```
cout << "Adding 3 and 4 gives us: " << 3 + 4;
```

在上面的示例程序中，程序中多次调用了add函数，但注意，我们并没有一遍遍地复制加法代码，而是多次调用封装了加法功能的add函数。如果函数比较短，调用函数没有多大帮助，但如果我们把更多的代码添加到add函数中（比如某些输出参数和结果的调试语句），调用函数将使得代码变动量很少——你仅仅需要修改函数，而不是修改所有重复的代码。

6.2　局部变量和全局变量

现在可以编写多个函数，每个函数可以有多个变量。接下来让我们花点时间讨论一下变量的名称。当在函数里声明一个变量时，你会对它命名。那么在哪些地方我们可以通过变量名引用变量呢?

6.2.1　局部变量

分析一个简单的函数：

```
int addTen (int x)
{
    int result = x + 10;
    return result;
}
```

函数中有x和result两个变量。首先讨论result，result只在定义它的括号内有效，即只对add函数内部的两行代码有效。换句话说，你可以在其他函数中使用result变量：

```
int getValueTen ()
{
    int result = 10;
    return result;
}
```

你甚至可以在addTen中调用getValueTen：

```
int addTen (int x)
{
    int result = x + getValueTen();
    return result;
}
```

上面有两个名为result的不同变量，一个属于addTen函数，另一个属于getValueTen函数。两个变量并不冲突，getValueTen执行时只会使用result变量的副本，addTen也是一样。

一个变量的有效范围称作它的*作用域*。变量的作用域指可以通过变量名称引用变量的区域。在函数内部声明的变量只在该函数内部有效。当主函数调用子函数时，主函数内声明的变量在子函数内无效，子函数内声明的变量也只在该子函数内部有效。

函数的参数在函数内部声明。尽管参数的值由调用的函数进行赋值，但这些参数对调用的函数无效。比如addTen函数中的变量x，它是函数的参数，只能在定义它的addTen函数中使用。此外，像其他在函数中声明的非参数类变量一样，参数变量x也不能被addTen函数的子函数使用。示例中，addTen的变量x对getValueTen函数无效。

函数参数就像传递给函数的变量的替身；改变函数参数对原始变量没有影响。当变量传递给函数时，变量的值被复制给函数参数。

```
#include <iostream>

using namespace std;
void changeArgument (int x)
{
    x = x + 5;
}

int main()
{
    int y = 4;
    changeArgument( y ); // y 值不改变
    cout << y; // 仍然输出4
}
```

示例代码21：local_variable.cpp

变量的作用域可以比函数代码区域小。C++用一组花括号定义小范围作用域。例如：

```
int divide (int numerator, int denominator)
{
    if ( 0 == denominator )
    {
        int result = 0;
        return result;
    }
    int result = numerator / denominator;
    return result;
}
```

第一个`result`的作用域只在`if`语句的花括号中，第二个`result`的作用域是从声明处到函数结尾。一般来说，编译器不会阻止你创建两个同名变量。在示例函数`divide`函数中，相似作用域下的多个同名变量会让看代码的人头疼不已。

在函数内部或代码块中声明的变量叫做*局部变量*。此外还有一种作用域更广的变量，叫做全局变量。

6.2.2　全局变量

有时，你可能想要某个变量对所有函数都有效。比如在棋盘游戏中，可能想把“棋盘”存储为一个全局性的变量，这样就可以让多个函数使用它而不需要次次都通过参数传值。

全局变量可以帮你实现这个功能。*全局变量*是一个声明在所有函数之外的变量，它在程序中声明代码后的任何地方都有效。

下面是一个如何声明和使用全局变量的基本示例。

```
#include <iostream>

using namespace std;

int doStuff () // 证明作用域的小函数
{
    return 2 + 3;
}

//全局变量可以像其他变量一样声明
int count_of_function_calls = 0;
void fun ()
{
    //全局变量再次有效
    count_of_function_calls++;
}
int main ()
{
    fun();
    fun();
    fun();
    //全局变量依然有效!
    cout << "Function fun was called " << count_of_function_calls << " times";
}
```

示例代码22：global_variable.cpp

变量count_of_function_calls的作用域从fun函数前开始。函数doStuff在它之前声明，所以不能使用它。fun和main在它之后声明，可以使用它。

6.2.3　有关全局变量的警告

全局变量似乎能让事情变得更容易，所有人都可以使用它。但是，使用全局变量会增加代码的阅读难度：想知道某个全局变量是否被使用过需阅读所有的代码！正确的做法是少用全局变量。只有当你确定有些事情需要大范围有效时才使用全局变量。否则请最好采用将参数传递给函数的办法，别让它们访问全局变量。即使你觉得某个特定的东西需要全局使用，但随后事实会证明没那么需要。

以前面的棋盘游戏为例，你可能计划写一个展示棋盘函数，通过访问全局变量实现。但如果你不想显示当前棋盘而想展示其他棋盘呢？例如，展示采用其他步法之后的棋盘。你写的这个函数不能将棋盘作参数，访问全局变量它就只能展示全局棋盘。这就不是很方便了。

6.3 使函数对调用有效

变量的作用域规则——只在声明后的区域有效，同样适用于函数（是不是很一致？）

例如，下面的程序不会成功编译：

错误代码

```
#include <iostream> // 需要cout

using namespace std;

int main ()
{
    int result = add( 1, 2 );
    cout << "The result is: " << result << '\n';
    cout << "Adding 3 and 4 gives us: " << add( 3, 4 );
}

int add (int x, int y)
{
    return x + y;
}
```

示例代码23：badcode.cpp

如果试图编译这个程序，你会看到如下错误信息（或类似信息）：

```
badcode.cpp:7: error: 'add' was not declared in this scope
```

问题出在调用add函数前它还没被声明，调用代码不在它的作用域中。调用一个未声明的函数会让编译器不解——它很无奈。

一种解决方案（我在示例中用过）是把整个函数放在调用它之前。另一种是在定义函数之前先进行声明。

尽管声明函数和定义函数听起来非常相似，但它们有着本质的区别。接下来详细解释这些术语。

6.3.1 函数定义和声明

定义一个函数意味着要给出完整的函数，包括函数体。例如，我们编写的add函数便是函数定义，因为它包含了add的功能。函数定义包含函数声明，因为函数定义需要用到所有函数声明提供的信息。

声明一个函数仅仅给出调用者需要的基本信息：名称、返回值类型和参数。函数在被调用之前必须先声明，不管是用函数声明还是给出完整的函数定义。

声明一个函数需要编写函数原型。声明将告诉编译器函数会返回什么，被谁调用以及它所传递的参数。你可以认为函数原型是函数使用指导。

```
Return_type function_name (arg_type arg1, ..., arg_type argN);
```

arg_type只表示每个参数的类型，如int、double或者char。这和声明变量是一回事。

下面是一个函数原型：

```
int add (int x, int y);
```

原型表明add函数有两个整型参数，也会返回一个整型数。分号告诉编译器这只是一个函数原型而不是完整的函数定义；不要忘记结尾的分号，免得编译出错。

6.3.2　函数原型的应用示例

下面是一个上面丢失函数原型代码的修正版本。

```
#include<iostream>

using namespace std;

// add的函数原型
int add (int x, int y);

int main ()
{
    int result = add( 1, 2 );
    cout << "The result is: " << result << '\n';
    cout << "Adding 3 and 4 gives us: " << add( 3, 4 );
}

int add (int x, int y)
{
    return x + y;
}
```
示例代码24：function_prototype.cpp

照常，程序由必须的头文件和using namespace std;开始。

接下来是以分号结尾的函数的声明。在这之后，包括main在内的所有代码都可以使用add函数，尽管add是在main之后定义。因为在main之前声明了原型，编译器根据声明能解析出它的参数和返回值。

谨记尽管函数可以在定义之前被调用，但最终（编译前）程序中必须包含函数定义。[①]

① 准确地讲，链接步骤会失败；我们会在之后讲解编译和链接之间的差异。

6.4　把程序拆分成函数

现在已经知道如何编写函数，你还需要知道什么时候需要编写函数。

6.4.1　当需要重复代码时

使用函数的主要目的是复用代码。函数可以让程序的部分逻辑复用起来更容易，当你想使用这些逻辑时，只需要调用函数即可，无需复制粘贴代码。复制粘贴尽管看起来也简单，但会导致代码重复很多次。使用函数可以节省代码空间，使程序易读易改。你会愿意在一个大型程序中修改40多次而不是只修改一个函数吗？反正我不会。

一个很好的经验是一旦你重复某些代码三次，就把这些代码封装成函数，方便以后重复使用。

6.4.2　使代码更加易读

即使无需复用代码，有一长段专业又复杂的代码也会让人很难理解你的代码。这时你可以编写一个函数并标上“这是我想用的功能”，然后使用这个功能即可。例如，如果你专门写一个函数处理“读取用户的输入”，它的功能很容易理解。否则要实现这个功能，你要编写代码处理索引按键，将按键转化为电信号，再对变量赋值，这很复杂！下面的写法会漂亮很多：

```
int x;
cin >> x;
```

阅读这个代码比阅读处理所有细节的代码好多了。当需要处理大量代码，你会发现很难抓住要点，此时需要编写一些函数来组织代码。

通过编写函数，你可以将注意力集中在函数的输入输出，而不是时时刻刻记住函数运行的细节。

你可能会想“难道我不需要知道细节吗”，当然需要，时常需要了解它所有的细节，但只需要查看某个函数即可，因为它所有的信息都在这里。当函数细节和程序结构混合在一起时，代码会很难阅读。

举一个菜单程序的例子，当用户选择一个菜单选项时，程序要运行复杂的代码。此时每个菜单选项应该对应一个函数。每个菜单项都可以通过查看对应的函数进行理解，主输入代码的结构也会容易理解。糟糕的代码通常只有基本的`main`函数，`main`函数里填充了大量乱七八糟的代码。事实上，下一章你会看到这种程序的例子。

6.5　命名和重载函数

为代码的变量、函数等选一个好名字是一件非常重要的事情，名字有助于理解代码。函数调

用不会展示函数实现的细节，挑选一个能够描述函数重要特征的名字非常重要。名字如此重要以至于有时候你想用同一个名字代表多个东西，例如通过三个坐标点计算三角形面积的函数：

```
int computeTriangleArea (int x1, int y1, int x2, int y2, int x3, int y3);
```

但还有一个通过长和高计算三角形面积的函数。你可能想再次使用名称computeTriangleArea，因为这个名字可以准确描述函数的作用。这会不会和之前的computeTriangleArea冲突呢？在C++中不会！C++允许函数重载；只要函数有不同的参数列表，多个函数可以共用一个名称。如下所示：

```
int computeTriangleArea (int x1, int y1, int x2, int y2, int x3, int y3);
```

和：

```
int computeTriangleArea (int width, int height);
```

编译器可以根据调用地址不同区分这两个函数调用，因为两个函数的参数数量不同。（编译器也能处理相同数量不同类型的参数。）所以像下面这两个函数：

```
computeTriangleArea( 1, 1, 1, 4, 1, 9 );
computeTriangleArea( 5, 10 );
```

编译器也能知道调用哪个函数。

重载函数不能滥用，两个有相同名字的函数并不意味着有一样的功能，但是如果两个函数参数不同但功能相同那么使用重载将比较有意义。

6.6　函数概述

和变量、循环、if语句一样，函数是C++程序员的基本工具。它可以在简单的接口下隐藏复杂的运算，处理重复的代码。这让以后复用代码更为方便。

6.7　问答题

(1) 哪个不是正确的原型？

A. `int funct(char x, char y);`　　B. `double funct(char x)`
C. `void funct();`　　D. `char x();`

(2) 函数原型`int func(char x, double v, float t);`的返回值类型是什么？

A. `char`　　B. `int`　　C. `float`　　D. `double`

(3) 下面哪个函数调用是有效的（假设函数存在）？

A. `funct;` B. `funct x, y;` C. `funct();` D. `int funct();`

(4) 下面哪个是完整的函数？

A. `int funct();`
B. `int funct(int x) {return x=x+1;}`
C. `void funct(int) {cout << "Hello"}`
D. `void funct(x) {cout << "Hello";}`

6.8 实践题

(1) 将之前编写的“菜单程序”改写成一系列的函数调用。每个菜单选项对应一个函数。增加计算器和“100 Bottles of Beer”两个可以被调用的函数。

(2) 将计算器程序中的每个类型的计算分解成单独的函数。

(3) 修改之前的密码程序，将密码检查逻辑从代码中分离，放入单独的函数中。

第 7 章

如何解决问题

7

现在已经学习了很多基础语言特性，估计你正在疯狂编写运行程序。但问题是，你怎么知道需要编写什么呢？即使了解当前问题，你依然会觉得自己像著名美国讽刺喜剧《南方公园》中的“内裤精灵”：

第一步：收集内裤
第二步：？？
第三步：获利

你知道了结尾和开头，却不知道过程。

在阅读代码时这一步被略过了，但是当自己编写程序时，便会遇到这个问题。（当然在某些情况下可能不会，这是一个好消息，你早于计划表；休息一晚，喝点啤酒，我们明天再见。）

OK，如果你对第二步比较迷糊，也没有什么关系。这部分非常有趣（别告诉你喝啤酒的那个小伙伴，他会后悔的）。

当然我也承认这是编程中最具有挑战性的部分，其难度比语法大。但这也是最令人满意的一部分。设计一个听起来很困难的东西，从草稿开始这是一件很神奇的事情；没有什么东西能比得上赋予程序生命，把困难的事情变得简单更棒的了。练习的越多，经验越多，但是首先你需要了解应该练习什么。这便是这章中要讲的内容。有个坏消息是第二步很可能会变成22步，因为解决问题的关键是把大问题分解成很多小问题。

让我们拿出刀具、材料开始做开胃菜。首先要对如何解决问题有一个基本的了解。当有一个绝妙的想法但不确定如何将其转换为代码时，你需要事先了解一些算法的基本概念。算法是解决问题的一系列步骤。即便你了解算法，依然不容易将逻辑转化为代码。或许程序需要实现的内容非常的多。幸运的是，有一些工具可以解决这个问题。

还记得我之前所说的编程是把内容分解成电脑可以理解的碎片吗？函数的优点就是能构建计算机可以理解的代码块，而不用一直处理原始信息。举个例子，如果想输出从1到100之间的素数，肯定要用到多个操作符，所以我们需要将其分解成电脑可以理解的步骤。

完成这个任务比较麻烦的地方在于需要做很多事情。在同一时间思考整个事情是相当艰巨的。

我们换一种思路：将其化整为零。每一步不需要单独的指令；只要尝试找到比目前方法更简单的方法即可。合理的步骤如下：

(1) 遍历从1到100的所有数字；
(2) 检查每个数字是不是素数；
(3) 如果是素数，输出。

OK，我们将其分解成一些不同的小问题，但是很明显无法将其转换成程序。还缺少什么呢？能否找到遍历1到100的数的方法？这听起来非常像一个循环。事实上，几乎可以实现这段代码了：

```
for ( int i = 0; i < 100; i++ )
{
    // 判断i是不是素数？如果是，将其输出。
}
```

在代码中放置一个占位符函数——isPrime。如果函数接收的参数是素数则返回true，否则返回false。接下来需要实现isPrime，假设函数存在，我们可以填写一部分代码。大部分函数我们可以思考、编写，可以把问题分解，判断一个数字是否是素数的难度比判断100个要小，所以思路非常正确。


```
for ( int i = 0; i < 100; i++ )
{
    if ( isPrime( i ) )
    {
        cout << i << endl;
    }
}
```

是不是很漂亮？我们已经有了一个基本的结构。现在唯一要做的是实现isPrime。让我们思考如何判断一个数是不是素数。素数是指除了1和此整数自身外，不能被其他数整除的数。这个定义给了我们足够的信息来把这个问题分解成更小的子问题。想判断一个数是否有除数，我们需要判断有没有数（除去1和自身）能将其整除。因为通过很多不同的数判断除数，所以需要另外一个循环。以下是这部分算法的具体步骤。

(1) 遍历从1到当前值。
(2) 如果被检测值能够被变量a整除，返回false。
(3) 如果不能被任何数整除，返回true。

来看看能否将以上内容转化为源代码。目前还不知道如何判断一个值能否被其他数整除，但是要坚定信念，假设我们可以实现，所以用一个isDivisible函数作为逻辑的占位符。

```
bool isPrime (int num)
{
```

```
    for ( int i = 2; i < num; i++)
    {
        if ( isDivisible( num, i ) )
        {
            return false;
        }
    }
    return true;
}
```

我们再一次把对一系列值的判断放到一个循环中。我们也把逻辑中的if语句翻译到代码中。

现在该如何实现isDivisible？一种方法是使用一个名为模运算符的特殊操作符，用符号%代替，返回整除的余数①。

```
10 % 2 == 0 // 10 / 2 = 5 没有余数
```

7.1　只需判断数被除时有无余数

```
bool isDivisible (int number, int divisor)
{
    return num % divisor == 0;
}
```

至此，我们已经把问题分解到电脑可以理解的地步。已经不需要编写任何其他函数了；程序中所有的代码要么是已经定义的指令，要么是我们定义的函数。把所有的放到一起：

```
#include <iostream>

// 注意函数原型的使用
bool isDivisible (int number, int divisor);
bool isPrime (int number);

using namespace std;

int main ()
{
    for ( int i = 0; i < 100; i++ )
    {
        if ( isPrime( i ) )
        {
            cout << i << endl;
        }
    }
}

bool isPrime (int number)
```

① 我提出这个新的操作符似乎有点不可思议，事实上有其他的方法判断一个数是不是有余数；我用模运算是因为这个最直接，如果你想做个练习，可以尝试找出同一个问题的不同解决方法。

```
{
    for ( int i = 2; i < number; i++)
    {
        if ( isDivisible( number, i ) )
        {
            return false;
        }
    }
    return true;
}

bool isDivisible (int number, int divisor)
{
    return number % divisor == 0;
}
```

通过使用函数原型，我们可以准确的执行一开始设想的代码。此外还可以像我们的设计一样从整体阅读代码，从辅助函数阅读函数内容。

7.2 效率和安全的简单说明

顺便聊几句，我们可以对代码进行改进使其更有效率，因为不需要在`isPrime`函数的循环中遍历2到`number`。最容易想到的算法并不意味着是最优最有效率的算法。此例中，我们要计算从2到`number`的平方。因为只计算了很小的一部分数字的素性，效率并不重要。然而，通常用在银行或电子商务网站进行敏感数据保护的RSA算法，需要产生大素数来创建加密密钥[1]。产生大素数便需要检查数字是不是素数。如果想产生大量RSA加密密钥，你需要一个快速、效率的素数生成器。

当遇到看起来很大很难解决的问题时，分解成小问题会更加容易管理。你不需要立即知道如何解决这些小问题（当然，这对如何解决并没有影响），只需要关心这些小问题的输入是什么，结果是什么。如果你能编写程序解决这些问题，就可以接受下一个挑战：实现这些小问题。专注一段时间，便能编写出源代码。

设计程序并不总是简单（如果是的话，便会有很多无聊的软件工程师了），有时候会因为一些原因无法解决子问题。当问题难以分解时，尝试后退一步，换一种可行的分解方案。

这种分解程序的方法叫做*自顶向下设计*，是一种强大的程序设计方法。另一种方法是*自底向上设计*，强调首先解决辅助函数，然后使用辅助函数解决大问题。自底而上设计可能会导致辅助函数完全用不到，但是从实现函数着手会有一个良好的开头。对于初学者来说，使用自顶向下设计会优于自底向上设计，因为这会帮助你专注于解决问题，尽可能准确找到所需要的辅助函数，

① 查看以下网址了解RSA算法：http://en.wikipedia.org/wiki/RSA_(algorithm) 。

而不是猜测哪些函数可能会用到[①]。

你无需用代码辅助设计。写在白纸或白板上可以让你观察前后是否合适，而无需关心C++语法和编译错误。如果直接用代码进行设计，在解决某些语法细节时可能会模糊大局。所以不直接编写代码，而是编写每一步过程并且把每个过程分解成更小的部分是正确且自然的设计的方法。

不过要注意一点，设计程序并不容易；我所告知的会有效，但是这并不是“万金油”。只有练习可以让你掌握，并且做得更好。可能会花费一些时间，不要放弃。

7.3　不知道算法的情况下的解决方案

在找素数的例子中，因为素数的定义几乎就是判断素数的算法，所以任务非常简单。最后只是把算法翻译成代码的问题。大多数情况下问题并不简单，你必须找到解决问题的算法。

例如，设想找出一种能将数字输出为英文名的算法（例如，输入1204，屏幕显示one thousand, two hundred four）。在交流时，这种转变会非常自然，无需设想算法的结构；只需要说即可。（假设英语是母语，如果不是，在解决这个问时你会有优势！）为了解决这类问题，你需要了解数据模型，才能想出算法。

编写几个示例并分析其相近或不同的地方直到找出模型是一个非常好的开始，如下所示：[②]

1	one
10	ten
101	one hundred one
1 001	one thousand one
10 001	ten thousand one
100 001	one hundred thousand one
1 000 001	one million one
10 000 001	ten million one
100 000 001	one hundred million one

看到模型的规律了吗？

1	**one**
10	**ten**
101	**one hundred** one
1 001	**one** thousand one
10 001	**ten** thousand one

① 请不要放弃尝试自底向上设计，它适合一些人，也可能适合你；如果无法理解自顶向下设计，在放弃之前反着想一下。

② 因为原文中是对英语进行转换，所以以下对照表都为英文，具体数字对应可自行搜索。——译者注

（续）

100 001	**one hundred** thousand one
1 000 001	**one** million one
10 000 001	**ten** million one
100 000 001	**one hundred** million one

每三个数字提升一层级，从无到thousand，到million。此外，对每个三数字组有“one, ten, one hundred”模型。然后用“高层级”进行合并：“one thousand”“ten thousand”和“one hundred thousand”。

这里的算法需要从把数字分解成三数字组开始，找到当前层级（thousand、million、billion）的“量级”然后把当前层级翻译成文本并与“量级”合并。每个三数字组都小于one thousand，所以需要解决更小的问题。继续观察更多的模型：

5	five
15	fifteen
25	twenty five
35	thirty five
45	forty five
105	one hundred five
115	one hundred fifteen
125	one hundred twenty five
135	one hundred thirty five
145	one hundred forty five

这里有一个相似的模式：如果有超过100的数字，文本便会是“X hundred”，接下来是两数字组的对应文本。如果没有百位，便只有两数字组。

接下来需要决定如何处理两数字组。你观察到这里仍是一个模型了吗？小于20的数，模式是各个数的对应，我们可以用一系列简单的`if-else`语句搞定。

处理1~19时不得不对程序进行硬编码——这里没有算法可以解决。这不是什么时候都能遇到的。

所以我们的算法如下。

(1) 分解数字到三数字组。

(2) 对每个三数字组，运算文本；追加组的量级；将组合并。

(3) 运算一个三数组的文本，计算百位的数字，把百位的数字转化成文本，添加hundreds，追加剩下两位的对应文本。

(4) 计算两数字组的文本，如果小于20，直接查找替换；如果大约20，运算十位的数字，查找单词，追加最后一位的数字。

我们需要把算法转化成源代码，不是所有的细节都清晰明了，但是你有足够的大纲帮助使用

自顶向下的设计方法实现算法。

你看到这个过程是怎样工作的了吗？通过比较不同的数字，我们可以发现数字构建的特定模型。能够找到算法的种子，虽然不是所有的细节都清晰，但是没有关系；我们会细分问题，直到问题解决。

7.4　实践题

(1) 完成将从−999 999~999 999的数字转换成英文文本的源代码。[①]

(2) 思考如何将英文文本转换成数字。这比之前的算法是难是易？如何处理错误输入？

(3) 设计算法找出从1到1000中质因子相加是质数的数（例如，12有质因子2、2和3，相加得7，是质数）。完成代码。[②]

① 你可能会用到整型能够截断小数点的功能。还有，记住算法无需适用于所有的数字，最多适用于6位数即可。
——译者注

② 提示：如果你不知道计算质因子的算法，可以上网搜索一下，我之前说过，程序员不需要了解数学。
——译者注

第 8 章

switch-case和枚举

到这里，我们先停止学习新的语言技巧，转而学习一点更基础的知识——选择性执行代码。（不是所有的东西都像函数和程序设计一样令人兴奋！）你经常会写一长串if-else语句来判断不同的变量，例如：在读取用户输入值时，需要判断值是多少；编写游戏时，需要判断按下的键是上键、下键、左键、右键还是空格键。本章中，我们将学习如何使用switch-case语句更方便的编写多条件判断代码，也将学习如何创建配合switch-case语句使用的变量类型。

如果单个变量需要和多个整数值进行比较，相对于使用多重嵌套的if语句，switch-case语句是一个非常好的替代品。一个整数值可表示为整数类型，比如int型或char型。

下面介绍switch-case语句的基本语法。switch后的变量在执行时将和每个case之后的值进行比较，当与其中的某个值匹配时，计算机将从匹配的case后的代码开始执行，直至switch-case模块结束，或遇到break语句。

```
switch ( <variable> )
{
case this-value:
    // 如果<variable>==this-value执行此处代码
    break;
case that-value:
    // 如果<variable>==that-value执行此处代码
    break;
// ...
default:
    // 如果<variable>不等于之后的任何case的值，执行此处代码
    break;
}
```

与给定变量值相等的首个case语句的冒号之后的代码将会执行。如果所有case语句中的值都不和给定变量值相等，程序将执行default case。default是可选语句，但最好包含它以处理意外case。

请注意每组case代码之后break的使用。break阻止程序顺势而下执行后面的case语句。不错，这个用法非常奇怪！但它的功能就在此，让程序有选择性地执行某个case代码。

每个case语句的变量必须是常量整型表达式，请注意下面这样是不合法的：

错误代码

```
int a = 10;
int b = 10;
switch ( a )
{
case b:
    // 代码
    break;
}
```

如果你尝试编译这段代码，将会看到如下的编译错误：

```
badcode.cpp:9: error: 'b' cannot appear in a constant-expression
```

下面是一个使用switch-case的示例程序：

```
#include <iostream>

using namespace std;

void playgame ()
{}

void loadgame ()
{}

void playmultiplayer ()
{}

int main ()
{
    int input;

    cout << "1. Play game\n";
    cout << "2. Load game\n";
    cout << "3. Play multiplayer\n";
    cout << "4. Exit\n";
    cout << "Selection: ";
    cin >> input;
    switch ( input )
    {
    case 1: // 注意这是冒号不是分号
        playgame();
        break;
    case 2:
        loadgame();
        break;
    case 3:
        playmultiplayer();
        break;
    case 4:
        cout << "Thank you for playing!\n";
        break;
```

```
        default: // 注意这是冒号不是分号
            cout << "Error, bad input, quitting\n";
            break;
        }
    }
```

示例代码25：switch.cpp

程序编译通过后将演示依据用户不同输入给出不同输出的简单模型，这段程序实现的功能非常像游戏《等待戈多》（*Waiting for Godot*）。

你可能会注意到一个问题，用户在程序结束前只能进行一次选择，如果用户输错了值，就没有再选择的机会。为了实现再选择，你可以把整个switch-case代码放入一个循环中，但那些break语句该怎么处理？它们会让循环退出吗？当然不会，break语句只会让代码跳转到switch语句的最后。

8.1 比较 switch-case 和 if-else

假设难以理解switch语句的逻辑，那么你可以尝试用if语句代替每个case语句，它们在本质上相同：


```
if ( 1 == input )
{
    playgame();
}
else if ( 2 == input )
{
    loadgame();
}
else if ( 3 == input )
{
    playmultiplayer();
}
else if ( 4 == input )
{
    cout << "Thank you for playing!\n";
}
else
{
    cout << "Error, bad input, quitting\n";
}
```

如果能用if-else做同样的事情，为什么还要用switch呢？switch的主要优点是它利用单个变量控制代码路径，可以清晰地显示程序工作流。而大量使用if-else语句时，每个变量都需要仔细阅读。

8.2　使用枚举创建简单类型

在编程中，有时你想让变量只代表某几个指定的值，且这些值事先已经确定。例如供用户选择的背景色是一组固定的值。如果能找到一种变量类型可以代表这一系列的常量将非常方便。此外，这类变量会非常适用于switch-case，因为变量代表的每个值都是已知的。

接下来我们学习枚举变量：enum。enum是“enumerated type”的缩写，是一系列固定值组合成的新的变量类型。彩虹颜色就是一个很好的枚举类型：

```
enum RainbowColor {
    RC_RED, RC_ORANGE, RC_YELLOW, RC_GREEN, RC_BLUE, RC_INDIGO, RC_VIOLET
};
```

要点：

(1) 关键字enum用来声明一个新的枚举；
(2) 每个新的枚举类型有自己的名字，RainbowColor；
(3) 类型中所有可能的值都被列举出来（我使用前缀RC_以防别人因为某些原因在其他的enum中使用同样的颜色名）；
4) 最后，别忘了分号。

现在你可以像下面这样初始化枚举变量RainbowColor：

```
RainbowColor chosen_color = RC_RED;
```

接着，可以编写如下代码：

```
switch (chosen_color)
{
case RC_RED: /* 红色 */
case RC_ORANGE: /* 橙色 */
case RC_YELLOW: /* 黄色 */
case RC_GREEN: /* 绿色 */
case RC_BLUE: /* 蓝色 */
case RC_INDIGO: /* 靛蓝色 */
case RC_VIOLET: /* 紫色 */
default: /* 处理错误类型 */
}
```

使用枚举类型，我们可以确定覆盖了变量的所有可能值。虽然变量类型的本质是整型，也可以接收枚举之外的值，但我不建议你用它表示其他的值，这会给程序维护人员带来麻烦。

你可能会奇怪：枚举到底是什么值？如果在声明枚举时没有提供特殊的值，那么值便是上一个枚举值加1，首个枚举的值是0。此例中，RC_RED是0，RC_ORANGE是1。

你也可以对枚举值进行自定义；如果代码需要使用来自另一个系统的特定值，比如一块硬件

或某些想取个好名字的需要复用的代码，此时自定义将会非常有用。

```
enum RainbowColor {
    RC_RED = 1, RC_ORANGE = 3, RC_YELLOW = 5, RC_GREEN = 7, RC_BLUE = 9,
RC_INDIGO = 11, RC_VIOLET = 13
};
```

枚举用处很大的一个主要原因是枚举允许对硬编码到程序中的值命名。例如，如果想写一个井字棋游戏，你需要找到表示棋盘上X和O的方法。你可能使用0代表空格，1代表O，2代表X。如果这么表示，你需要编写一些代码将棋盘上的每个空格与0、1和2比较。

```
if ( board_position == 1 )
{
    /* 因为是O，在此处执行一些操作 */
}
```

这样的代码难以阅读，因为代码中的幻数有特殊含义，仅通过阅读代码很难理解这些数字意味着什么（除非代码中有注释），而enum可以让你对这些值命名:

```
enum TicTacToeSquare { TTTS_BLANK, TTTS_O, TTTS_X };

if ( board_position == TTTS_O )
{
    /* 代码 */
}
```

只有当未来某些可怜虫要修补bug时（这个可怜虫可能是你！），通读代码才能理解程序要做什么。

枚举比较适合处理固定类型的输入，switch-case比较适合处理用户输入，但这两种语句都不适合处理大量输入数据。例如，你可能想读取一大堆棒球或足球的统计数据进行处理。这种情况下，你需要的不是switch-case，而是能存储并处理大量数据的方法。

这些是本书第二部分将要提及的。在此之前，我们会多了解一些方法，在不需要处理大量数据时编写程序做一些新鲜有趣的事情。具体来讲，接下来我们将学习随机数的使用（做游戏时可能会用到）。

8.3　问答题

(1) 紧接着case语句的是什么？

A. :　　　B. ;　　　C. -　　　D. 新的一行代码

(2) 避免从一个case执行到另一个case需要什么？

A.end;　　　B. break;　　　C. Stop;　　　D. 分号

(3) 哪个关键字用来处理未知情况？

A. `all`　　B. `contingency`　　C. `default`　　D. `other`

(4) 下面代码的运行结果是？

```
int x = 0;
switch( x )
{
    case 1: cout << "One";
    case 0: cout << "Zero";
    case 2: cout << "Hello World";
}
```

A. `One`　　B. `Zero`　　C. `Hello World`　　D. `ZeroHello World`

8.4　实践题

(1) 用`switch-case`重写第6章中的“菜单程序”。

(2) 使用`switch-case`编写程序输出The Twelve Days of Christmas[1]的所有歌词。（提示：你可能想要`case`语句顺序执行，那么请别用`break`语句。）

(3) 编写两个玩家的井字棋游戏，允许两人对抗；使用枚举代表棋盘的值。

① 参见http://en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)。

第9章 随机

9

有两种方法可以让你的程序每次运行结果不同：

(1) 让用户输入不同的数据（或者从文件中读取不同的数据）；
(2) 对用户输入的相同数据采取不同的处理方式，使其运行结果不同。

大多数情况下，第一种方法是非常好的，用户总是希望他们程序的结果是可预测的。比如当编写一个文本编辑器或者网页浏览器时，你会希望程序在用户每次输入一段文本或网址时执行同样的操作，而不是由浏览器随机决定访问哪个页面，除非是使用StumbleUpon[①]。

但在某些情况，每次执行相同操作并不是一个好的处理方式。例如，很多电脑游戏依赖随机，俄罗斯方块便是一个典型的例子，如果每次游戏方块的下落顺序都相同，用户便会记住下落顺序，因为可以预测接下来会出现什么方块，所以得分会一次比一次高。最后游戏和背诵圆周率的千位小数没啥不同。为了让俄罗斯方块游戏更有意思，程序需要随机选择下一次方块的形状和朝向。

为了实现这个功能，计算机需要生成随机数。因为计算机会准确执行命令，当我们执行相同的操作时计算机总会返回同样的结果。这就很难生成真正的随机数。不过没有必要生成真的随机数。生成像随机数的数也能达到目的，这就是伪随机数。

要生成伪随机数，计算机需要一个种子，利用数学变换将种子转换成另一个值。新值再成为下一个种子。如果程序每次采用不同的种子，程序便永远不会生成相同的数据序列。这里使用的数学转换需要特别挑选，要让所有数字的生成概率相等但又不会有明显的计算模型。（例如，它不会只是每次对数字加1。）

C++提供了所有的功能。你无需关心数学转换，C++中有相关的函数实现。所有你要做的只是提供随机种子，使用当前时间作种子即可。让我们看一下细节：

① StumbleUpon是一个能让你“偶遇”有趣网页的网站：http://www.stumbleupon.com/。

9.1 获得随机数

C++有两个函数，一个是设置随机种子，另一个是用种子产生随机数：

```
void srand (int seed);
```

srand函数将某个数字设置为种子。在程序开头处需要调用一次srand。使用srand的典型方法是把time函数的结果作为参数，time函数返回一个代表当前时间的数值：[①]

```
srand ( time ( NULL ) );
```
[②]

如果连续调用srand，程序会反复地更新随机数发生器种子，因为连续调用的时间序列非常相近，生成的随机数也会很相近。使用srand必须包含cstdlib头文件，使用time函数必须包含ctime头文件。

```
#include <cstdlib>
#include <ctime>

int main ()
{
    //在最开始处调用一次
    srand( time( NULL ) );
}
```

示例代码26：srand.cpp

参照下面原型调用rand函数来获取随机数。

```
int rand ();
```

注意，rand函数没有任何参数，仅有一个返回值。让我们将返回值输出出来。

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main ()
{
    //在最开始处调用一次
    srand( time( NULL ) );
```

① time函数返回从1970年1月1日起到现在的秒数。这个规则源自于Unix操作系统，有时它称为**Unix time**。大多数情况下，时间存储在32位有符号整型中。随着时间的增加，秒数会超过整型可表示的范围，最后将以负数结尾表示过去的时间。超过整型数的现象将发生在2038年，它引起了对“2038年问题”（Year 2038 Problem）的讨论，使用Unix time的计算机程序将会把2038年当做1901年处理。详情请参考：http://en.wikipedia.org/wiki/Year_2038_problem。

② 目前你不用了解NULL参数，先就照着这么写；在后面关于指针的一章中会介绍很多关于它的内容。

```
    cout << rand() << '\n';
}
```

示例代码27：rand.cpp

太好了！程序每次运行结果都不同，这意味着你可以玩上好几小时的猜数字游戏！

不过，这游戏可能也不那么令人兴奋，毕竟数字的范围太大了。如果数字能设定在某个范围内，你可以做更有趣的事情。原本调用rand函数会返回从0到常量RAND_MAX之间的某个值（RAND_MAX至少是32767）。这个范围很大，你可能只需要其中的一小部分，有个解决方案是循环调用rand函数，直到它返回规定范围内的值。

```
int randRange (int low, int high)
{
    while ( 1 )
    {
        int rand_result = rand();
        if ( rand_result >= low && rand_result <= high )
        {
            return rand_result;
        }
    }
}
```

但这个办法非常糟糕！首先是慢，如果想获取1到4之间的数字，程序要花费漫长的时间来获取这小范围内的数字，因为rand返回值的范围实在太大。其次是无法保证程序会顺利结束，程序可能（尽管这不太可能）永远无法获取所需范围内的值。这儿有一个更好的方法使你不必冒这个险。

C++有一个返回除法余数的操作符（如4/3商为1，余数为1）——模数运算符。之前的几章中我们曾使用它判断质数。如果你没有注意到也不要紧，人们总是自动屏蔽数学函数。但模数非常有用。因为被4整除的余数的范围是0~3。如果用rand函数返回的随机数除以所需数字的范围长度（即范围内数的数量），便会获得0到最大范围之间的值（不包含最大值）。

例如：

```
#include <ctime>
#include <cstdlib>
#include <iostream>

using namespace std;

int randRange (int low, int high)
{
    //先获取随机数，再处理得到从0到所需数字范围长度的值，然后加上最小值
    return rand() % ( high - low + 1 ) + low;
}

int main ()
```

```
{
    srand( time( NULL ) );
    for ( int i = 0; i < 1000; ++i )
    {
        cout << randRange( 4, 10 ) << '\n';
    }
}
```

示例代码28：modulus.cpp

这段程序有两点需要注意的地方。首先，我们必须对high-low加1，举例说明原因，设想目标范围是0到10，当中有11种可能出现的值。减法获得的是两个值之间的差值，比范围内值的数量少1，因此必须加1。其次，注意我们需要加上目标范围的最小值，设想如果想获取10到20之间的数，通过上面的方法只能获取0到10之间的随机数，再加10才能将范围设定到10到20之间。

掌握获取特定范围的随机数后，我们可以编写一些类似《猜数字》(*guessing games*)或《模拟骰子》(*simulate dice rolls*)等有趣的程序。

9.2　bug 和随机数

随机数在程序开发阶段会带来不方便。如果程序有bug，你会希望程序每次运行的结果都是固定的，如果使用随机数，程序的bug可能不会每次都出现，这样就需要花很长时间测试运行程序并且要求得到的都是正确的结果，即使这样还可能出现意料之外的bug。当测试或调试程序时，你可以注释掉srand调用。因为当随机数发生器srand不更新种子时，rand函数在每次程序运行后会返回同样序列的值，如此一来程序每次执行的结果便会相同。

要是在调用srand后才出现bug该怎么办？一个技巧是记录程序每次运行的种子值：

```
int srand_seed = time( NULL );
cout << seed << '\n';
srand( srand_seed );
```

当发现bug时，你可以修改程序，使用相同的随机种子可以最快找到bug。例如，如果种子是35 434 333，可以修改程序为：

```
int srand_seed = 35434333; // time( NULL );
cout << seed << '\n';
srand( srand_seed );
```

这样程序每次运行都能得到可预期的值。

9.3　问答题

(1) 在rand之前不调用srand会发生什么？

A. rand失败
B. rand一直返回0
C. rand会在每次程序运行时返回同样的数列
D. 什么都不发生

(2) 为什么要用当前时间作为srand的种子?

A. 确保程序运行一致
B. 每次程序运行时产生新的随机值
C. 确保计算机产生真随机数
D. 这样做是为了方便你在对同一个操作需要多次设置种子时只调用一次srand即可完成

(3) Rand返回值的范围?

A.想多少就多少　　B. 0 到1000
C. 0 到RAND_MAX　　D. 1 到RAND_MAX

(4) 表达式11 % 3的结果是?

A. 33　　B. 3　　C. 8　　D. 2

(5) 什么时候应该使用srand?

A. 每次需要随机值时
B. 永远不需要，这只是Windows的一个封装
C. 一次，在程序的开头
D. 偶尔，在while循环内使用rand之后增加随机性

9.4　实践题

(1) 编写程序模拟抛硬币，运行多次查看结果是否随机。

(2) 编写程序选取1到100间的某个数字，让用户猜测。程序需要告诉用户猜测的数与正确数相比是大、小还是刚好。

(3) 编写程序升级问题(2)中的猜数游戏。限定用户猜测的次数?

(4) 编写《老虎机》(*slot machine*)游戏，对玩家随机展示老虎机的结果——每一轮有三种(或更多)值。不用担心显示文本“spinning”，你只需要随机选择结果展示并输出游戏的胜利者(由你自行设定奖金组合)。

(5) 编写《扑克牌》(*poker*)游戏，首先对玩家提供5张牌，接着让用户选择新牌，然后判定手上的牌是不是好牌。想想这个游戏是否容易实现。如果要记录已经出过的牌想想可能会遇到什么问题?与《老虎机》游戏相比这个游戏是简单还是困难?

Part 2 第二部分

数据处理

你已经学习了如何编写基础程序来实现一些有趣的东西，如显示输出（比如你的名字）、和用户交互、根据用户输入执行不同的操作、重复执行简单操作，甚至创建游戏。

这些都是好东西，但是一段时间之后，你可能会觉得这些程序很枯燥；只处理少量的数据没有什么乐趣。但目前为止，你还未学习如何处理大量数据。回想前一章的扑克牌练习题，追踪被使用的牌是否很简单？将整副牌洗牌并展示又会有多困难？

[短暂地停顿一下]

很难。首先需要能存储 52 个不同值的方法，这需要 52 个不同的变量。每次对新牌赋值时，你都需要检查每个变量，看自己是否已经绘制它所代表的牌。当处理第 52 张牌时，你会面对大量代码，没有心思编写更多的程序。幸好程序员都很懒，他们不喜欢做那些并不是非做不可的事，于是他们想出了非常好的方法来解决这个问题。

此部分内容是关于如何解决这些问题的，让你知道如何处理大量数据：读取、存储和操作。我们首先给出一个技巧，说明如何在不创建很多不同变量的情况下持有大量数据，这会帮助我们解决扑克牌问题。

第 10 章 数 组

数组是“如何简单存储大量数据”的答案，本质上是一个通过单个名字存储多个数据的变量，但每个数据都有一个数字索引。你可以认为数组是一个可以通过数字访问元素的编号列表。

数组很容易想象：

val0	val1	val2	val3	val4	val5

数组就像一串彼此相连的盒子；每个盒子是数组的一个*元素*。获取数组的值就像通过数字寻找特定的盒子：“给我5号盒子”，这是很神奇的事情，因为一个数组用一个名字存储所有的数据，很容易以*编程的方式*选择数组的元素。如果想绘制5张扑克牌，可以在大小为5的数组中存储5张卡片。当选取新的卡片时只要通过索引修改数组即可，不用使用新的变量。因此，在绘制每个独立的卡片时，使用变量存储索引可以让你使用同一段代码，而无需对每个变量编写不同的代码。不同之处如下：

```
Card1 = getRandomCard();
Card2 = getRandomCard();
Card3 = getRandomCard();
Card4 = getRandomCard();
Card5 = getRandomCard();
```

和：

```
for ( int i = 0; i < 5; i++ )
{
    card[ i ] = getRandomCard();
}
```

不妨设想一下如果有100种牌的情况！

10.1 数组的基础语法

声明数组需要指明两个内容（除了名字）：类型和大小。

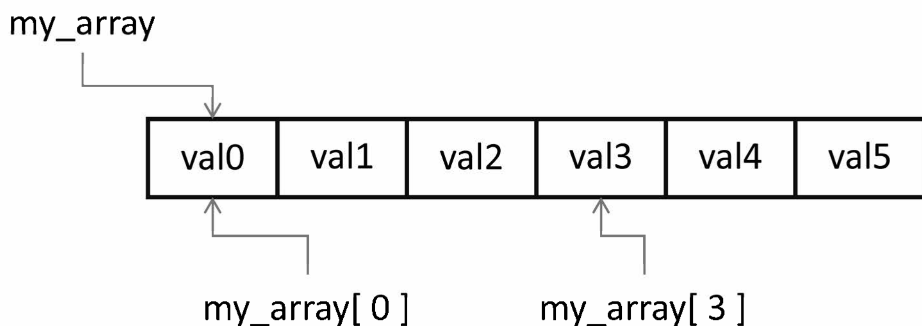
```
int my_array[ 6 ];
```

这句代码声明了一个包含6个整型元素的数组。注意变量名之后的方括号，以及方括号内的数组大小。

访问数组里的元素需要使用方括号，不过这次方括号内的不是数组大小，而是被访问元素的索引。

```
my_array[ 3 ];
```

我们可以这么想象一下：



my_array代表整个数组，但my_array[0]代表第一个元素，my_array[3]代表第四个。如果对此比较疑惑，说明你注意力比较集中。这不是排版错误，数组的索引从0开始。通过索引的意思是通过方括号内的数字获取数组中特定的值。你可能对此不习惯，除非父母（或教你识数的那个人）是程序员。

有一个简单的思考方法：索引是到达盒子前需要通过的队列的长度。你有可能在其他地方遇到过偏移量这个词。偏移量只是描述某些事情的独特方法，数组中的值相对于数组开始的偏移量是索引的大小。因为数组的第一个元素在开头，所以偏移量和索引是0。

当选取了数组中特定值后，可以像其他变量一样使用。修改数组中的元素如下方代码：

```
int my_array[ 4 ]; //声明数组
my_array[ 2 ] = 2; // 设置数组中的第三个元素（已经声明）为2
```

10.2 数组使用示例

10.2.1 使用数组存储排序

回忆之前的问题：“如何对52张牌洗牌？”现在有了数组这个存储方法，可以解决存储52张牌的问题。问题的另一部分是如何在面板上展示卡片的顺序。因为数组用数字的形式访问，所以

你可以把面板上牌的顺序对应于数组中的元素顺序。如果用52个不同的值随机对数组赋值，可以认为数组中的第一个元素（索引0）是面板的开头，最后一个元素（索引51）是底部。

数组的另一个常见用法是存储排序值。例如读取100个值并输出排序值。忽略排序的问题，表示值编号的方法是将值放入数组，并利用数组的原始顺序。

10.2.2　用多维数组表示网格

数组也可以用来表示多维数据，如象棋棋盘[或者简单一点的东西，如井字棋棋盘（tic-tac-toe board）]，多维数据的索引不止一个。

声明二维数组需要提供每维的大小：

```
int tic_tac_toe_board[ 3 ][ 3 ];
```

下面是井字棋棋盘tic_tac_toe_board的简单可视化：

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

因为二维数组是方形，所以访问元素时需要两个索引，一个代表横向、一个代表纵向。两个索引准确标记了表格中被访问元素的位置。访问元素只需要两个值，一个在第一个方括号内，另一个在第二个方括号内。

你可以创建三维数组，尽管可能不需要。事实上，你可以创建四维、五维或多维数组。这不好图示化，实践中也很少使用，所以我不进行演示。

一个网格型的数组可以让你更好地组织数据；对于井字棋棋盘，可以把数组中每个元素的值和棋盘位置对应。你还可以使用数组表示RPG（角色扮演游戏）游戏的迷宫或水平布局。

10.3　使用数组

10.3.1　数组和**for**循环

数组和for循环非常搭配；访问数组时可以使用将初始化为0的变量逐渐递增直到数组长度的模式。这种模式和for循环模型非常相符。

下面是一个演示使用for循环创建乘法表并用二维数组存储结果的小程序：

```
#include <iostream>

using namespace std;

int main ()
{
    int array[ 8 ][ 8 ]; // 声明像棋盘似的数组

    for ( int i = 0; i < 8; i++ )
    {
        for ( int j = 0; j < 8; j++ )
        {
            array[ i ][ j ] = i * j; // 对每个元素赋值
        }
    }
    cout << "Multiplication table:\n";
    for ( int i = 0; i < 8; i++ )
    {
        for ( int j = 0; j < 8; j++ )
        {
            cout << "[ "<< i <<" ][ "<< j <<" ] = ";
            cout << array[ i ][ j ] <<" ";
            cout << "\n";
        }
    }
}
```

示例代码29：multidimensional_array.cpp

10.3.2 将数组传递给函数

你会很快学到语言的交互特性。例如现在了解了数组，你会提问："怎么把数组传递给函数？"幸运的是，这里没有太多语法。

当调用函数时，你只需要使用数组的名字：

```
int values[ 10 ];
sum_array( values );
```

声明函数时，如下所示输入数组名：

```
int sum_array (int values[]);
```

"等下，"你可能会疑惑，"怎么回事？没有数组大小！"没错，对于一维数组，不需要指定大小。定义数组时需要指定大小，因为编译器需要开辟空间；但传递数组给函数时，仅仅传递了原来的数组，因为没有创建新的数组，所以不需要指定大小。事实上将原来的数组传递给函数意味着当你修改了函数内的数组时，发生的改变在函数结束后依然有效。而我们之前看到的那些普通

变量改变的是副本；当函数接收参数并修改变量时并不会影响原来的值。

当然，使用数组时，除非函数知道数组大小，否则函数需要将数组大小作为第二个参数：

```
int sumArray (int values[], int size)
{
    int sum = 0;
    for ( int i = 0; i < size; i++ )
    {
        sum += values[ i ];
    }
    return sum;
}
```

不过，传递多维数组时需要指定除首个之外的每个维度大小。

```
int check_tic_tac_toe (int board[][ 3 ]);
```

这非常奇怪！但目前你只需要记住无需包含首个大小（尽管可以填写，但是编译器会忽略）。

在介绍指针时我会详细介绍关于传递数组给函数的内容。那时我会解释发生在幕后的运算。现在只需要把它当成奇怪的语法即可。

让我们编写一个完整的程序演示sum_array函数：

```
#include <iostream>

using namespace std;

int sumArray (int values[], int size)
{
    int sum = 0;
    //当i==size时数组停止，因为数组的大小就是size
    for ( int i = 0; i < size; i++ )
    {
        sum += values[ i ];
    }
    return sum;
}

int main ()
{
    int values[ 10 ];
    for ( int i = 0; i < 10; i++ )
    {
        cout << "Enter value " << i << ": ";
        cin >> values[ i ];
    }
    cout << sumArray( values, 10 ) << endl;
}
```

示例代码30：sum_array.cpp

思考如何不使用数组编写这类程序。因为没有存储所有值的方法，你不得不保持连续相加——每当用户输入时，需要立即相加。当你想在后面使用数字时，很难跟踪所有的数字（例如，显示被相加的数字）。

10.3.3　注销数组的末尾

如果数据超过数组的大小，请一定不要尝试把数据写在数组的最后一个元素之外，比如你有一个10个元素的数组，然后尝试在偏移量`10`上写入数据。

错误代码

```
int my_array[ 10 ];
my_array[ 10 ] = 4; //尝试对第11个元素赋值
```

数组只有10个元素，所以最后一个有效的数组索引是`9`。使用索引`10`是无效的，有可能会引起程序崩溃！（讨论内存时会解释崩溃原因。）这种情况最可能的场景发生在编写循环遍历数组时：

错误代码

```
int vals[ 10 ];
for ( int i = 0; i <= 10; i++ )
{
    cin >> vals[ i ];
}
```

代码中数组有10个元素，但是循环条件会判断`i`小于等于`10`；也就是说这将会写数据到`vals[ 10 ]`，这是不应该的。而不幸的是，尽管编译器非常严格，但是却不会通知你这些bug。你只会知道当程序崩溃或异常时程序会出现问题，因为改变的值被其他的代码所使用。

10.4　数组排序

让我们尝试回答之前的问题：“如何接收100个值并对其排序？”代码的基本框架现在应该相当清楚了，首先你需要一个循环读取用户的100个整型值：

```
#include <iostream>

using namespace std;

int main ()
{
    int values[ 100 ];
    for ( int i = 0; i < 100; i++ )
    {
        cout << "Enter value " << i << ": ";
        cin >> values[ i ];
```

```
    }
}
```

示例代码31：read_ints.cpp

这是很简单的部分，现在你已经获取到了数据，然后怎么进行排序呢？大多数人首先想到的是找到列表中的最小值，移到最开始。然后查找列表中第二小的值，然后移动到第一个值的后面。然后查找列表中第三小的值，移动到第二个值后面。

如果对以下列表排序：

```
3，1，2
```

首先需要将1移动到列表的开始：

1，3，2

然后把2移动到列表的第二个位置上：

1，**2**，3

这样看起来你是不是可以使用之前学过的C++特性来编写代码了？看起来很像循环。遍历整个数组，从第一个元素开始，查找数组中剩下的元素中最小的值（数组中未排序的部分），判断是否可以放置在这里。然后和当前索引的值进行交换（当前值必须移到其他地方）。采用自顶向下的设计方法，开始编写一部分代码：

```
void sort (int array[])
{
    for ( int i = 0; i < 100; i++ )
    {
        int index = findSmallestRemainingElement( array, i );
        swap( array, i, index );
    }
}
```

现在我们可以考虑实现这两个辅助方法：`findSmallestRemainingElement`和`swap`。首先思考`findSmallestRemainingElement`；这个函数需要从索引`i`开始，遍历整个数组，找到数组中的最小的元素。这个听起来像不像另外一个循环？我们查找数组的最小值，如果小于当前最小的元素，便设定这个元素的索引为当前最小元素的索引。

```
int findSmallestRemainingElement (int array[], int index)
{
    int index_of_smallest_value = index;
    for (int i = index + 1; i < ???; i++)
    {
        if ( array[ i ] < array[ index_of_smallest_value ] )
        {
            index_of_smallest_value = i;
        }
```

```
    }
    return index_of_smallest_value;
}
```

这看起来是不是很合理?但是有一个小问题,什么时候循环应该停止呢?函数参数中没有信息指明数组的大小!我们需要添加上,调用函数findSmallestRemainingElement时同样需要大小。注意,自顶向下设计需要返回到上层源代码并做一些修改——这是设计程序的正常部分,不用担心。让我们做些修改使排序源代码无需把100硬编码进去。

```
int findSmallestRemainingElement (int array[], int size, int index)
{
    int index_of_smallest_value = index;
    for (int i = index + 1; i < size; i++)
    {
        if ( array[ i ] < array[ index_of_smallest_value ] )
        {
            index_of_smallest_value = i;
        }
    }
    return index_of_smallest_value;
}

void sort (int array[], int size)
{
    for ( int i = 0; i < size; i++ )
    {
        int index = findSmallestRemainingElement( array, size, i );
        swap( array, i, index );
    }
}
```


最后需要实现swap函数。因为函数可以修改传递进去的原始数组,所以需要使用临时变量保存被重写的第一个值:

```
void swap (int array[], int first_index, int second_index)
{
    int temp = array[ first_index ];
    array[ first_index ] = array[ second_index ];
    array[ second_index ] = temp;
}
```

因为传入swap函数的原始数组可以直接修改,这么实现就可以了。

你可以用随机数对数组赋值并对数组排序,证明这个排序算法是否能正常工作。下面是完整程序:

```
#include <cstdlib>
#include <ctime>
#include <iostream>
```

```
using namespace std;

int findSmallestRemainingElement (int array[], int size, int index);
void swap (int array[], int first_index, int second_index);

void sort (int array[], int size)
{
    for ( int i = 0; i < size; i++ )
    {
        int index = findSmallestRemainingElement( array, size, i );
        swap( array, i, index );
    }
}

int findSmallestRemainingElement (int array[], int size, int index)
{
    int index_of_smallest_value = index;
    for (int i = index + 1; i < size; i++)
    {
        if ( array[ i ] < array[ index_of_smallest_value ] )
        {
            index_of_smallest_value = i;
        }
    }
    return index_of_smallest_value;
}

void swap (int array[], int first_index, int second_index)
{
    int temp = array[ first_index ];
    array[ first_index ] = array[ second_index ];
    array[ second_index ] = temp;
}

// 显示排序前和排序后数组的辅助方法
void displayArray (int array[], int size)
{
    cout << "{";
    for ( int i = 0; i < size; i++ )
    {
        //格式化输出列表，如果不是第一个元素，便添加逗号
        if ( i != 0 )
        {
            cout << ", ";
        }
        cout << array[ i ];
    }
    cout << "}";
}

int main ()
{
    int array[ 10 ];
    srand( time( NULL ) );
```

```
    for ( int i = 0; i < 10; i++ )
    {
        //为方便阅读限制数字大小
        array[ i ] = rand() % 100;
    }
    cout << "Original array: ";
    displayArray( array, 10 );
    cout << '\n';

    sort( array, 10 );
    cout << "Sorted array: ";
    displayArray( array, 10 );
    cout << '\n';
}
```

示例代码32：insertion_sort.cpp

上面的排序算法称为插入排序，虽然不是排序算法中最快的一个，但它容易理解并实现。如果对于非常庞大的数据进行排序，需要选择更快但更难实现和理解的算法。作为程序员，你需要权衡这些。大多数情况下，越容易实现的算法越好，但是如果每天需要处理网站上数百万的用户数据，最简单的算法通常难以胜任。这需要你根据数据量和算法速度决定使用哪个算法。如果你是整晚跑一个批处理作业，慢一点没有问题，但是如果需要实时回复用户的搜索（如谷歌），这就不行了。

正如你所见，数组提供的很多处理能力让我们可以处理比以前多很多的数据。尽管仍有少量问题需要去解决。如果想关联多个不同但相关的值，而不是存储单一的值，该怎么办？数组可以帮助组织数据的不同部分，但是却不能组织不同数据类型的数据。在下面结构这一章中，我们会看到解决方法。

第二个问题是，数组提供的内存大小是固定的，在编写程序时已经固定。如果想处理不限数量的数据，固定大小的数组将不能胜任。我们同样会在后面的几章中解决这个问题。

尽管有这些限制，数组依然非常重要。使用索引访问数据的思想会随时随地出现。

10.5 问答题

(1) 下面的声明数组中哪个正确的？

A. `int anarray[ 10 ];`　　B. `int anarray;`
C. `anarray{ 10 };`　　D. `array anarray[ 10 ];`

(2) 有29个元素的数组的最后一个元素的索引是什么？

A. `29`　　B. `28`　　C. `0`　　D. 程序定义的索引

(3) 下面哪个是多维数组？

A. `array anarray[ 20 ][ 20 ];`　B. `int anarray[ 20 ][ 20 ];`
C. `int array[ 20, 20 ];`　D. `char array[ 20 ];`

(4) 一个有100个元素的数组`foo`，下面哪个可以正确访问第7个元素？

A. `foo[ 6 ];`　B. `foo[ 7 ];`　C. `foo( 7 );`　D. `foo;`

(5) 下面哪个函数可以接收二维数组？

A. `int func ( int x[][] );`　B. `int func ( int x[ 10 ][] );`
C. `int func ( int x[] );`　D. `int func ( int x[][ 10 ] );`

10.6　实践题

(1) 编写代码实现插入排序的函数，可以处理任意大小的数组。

(2) 编写程序读取50个值，输出最大值、最小值、平均值以及50个输入值，每行输出一个。

(3) 编写程序检测数组是否已经排序，如果没有，进行排序。

(4) 编写两个玩家对战的井字棋游戏。程序可以判断玩家胜利或棋盘被填满（平局）。额外任务：你能否让程序在和棋之前不让任何一方赢？

(5) 编写井字棋游戏，棋盘大小超过3×3。四点一线即赢。在程序开始时允许玩家设置棋盘大小。（提示：目前在编译时只能定义棋盘大小为固定值，所以你可以限制棋盘的最大值。）

(6) 编写两人跳棋，允许玩家移动，判断移动是否合法和游戏是否结束。必须支持国王！随便添加任何规则。允许用户在程序启动时选择游戏规则。

第 11 章

结构体

11.1 关联多个值

现在可以将单个值存储在一个数组中了，这使你有可能编写出可以处理大量数据的程序。当你处理更多的数据时，可能会遇到不同数据块间有关联的情况。比如，你想将电子游戏中多个玩家在屏幕上的坐标（*x*和*y*值）与玩家的名字一起存储。现在可以用三个独立的数组来实现这个目的：

```
int x_coordinates[10];
int y_coordinates[10];
string names[10];
```

但要注意，每个数组都是跟其他数组相关联的。因此，如果你移动了某一个数组里某个元素的位置，就必须移动另外两个与其相对应的数组元素的位置。

又要跟踪第四个数组里的值时，这会变得非常烦琐。你必须增加另一个数组，并使其与原来的三个数组保持同步。幸好设计编程语言的人不是受虐狂，设计了一个更好的能够将相关联的值组合起来的方法——结构体。结构体允许你将不同的值存储在同一个变量名下的不同变量中。当多块数据需要组合在一起时，结构体就能派上用场了。

11.1.1 语法

定义一个结构体的语法格式是：

```
struct SpaceShip
{
    int x_coordinate;
    int y_coordinate;
    string name;
}; // <- 注意分号，千万不能漏掉它
```

此处的SpaceShip是我们所定义的特定结构体类型的名称。换句话说，你创建了自己的类型。

就像使用double或int一样，你可以使用这个结构体类型来声明一个变量：

```
SpaceShip my_ship;
```

变量名x_coordinate、y_coordinate和name都是新类型的域（field）。等等，域，什么是域?

实际情况是这样的：刚刚我们创建了一个复合类型，它是一个存储了多个相互关联的值（比如屏幕的*x*和*y*坐标，或姓氏与名字）的变量。通过把要访问的值命名为域这种方式，可以区别这个复合类型变量中的值。这就像是两个不同名的变量，但不同的是，这两个变量被组合在了一起，并以一致的方式来命名。你可以把结构体想象成具有多个域的表单（想一想驾驶执照应用程序），表单里存储了大量的数据，表单的每个域就是一块特定的相关数据。声明一个结构体等价于定义一个表单，声明一个该结构体类型的变量等价于创建一个该表单的副本，可以用来填写和存储一连串的数据。

要访问结构体的域，将“.”加在结构体变量名后。（注意，不是结构体类型名后，每个结构体变量都有自己独立的域和值。）接着，写域的名字：

```
// 声明变量
SpaceShip my_ship

// 使用该变量
my_ship.x_coordinate = 40;
my_ship.y_coordinate = 40;
my_ship.name = “USS Enterprise (NCC - 1701 - D)”;
```

如你所见，一个结构体里可以有许多域，这些域没有数量限制，也不要求类型相同。

现在，我们来看一个示例程序，该程序将结合数组和结构体，演示一个游戏读取5位玩家名称的过程（不包括该游戏的主体程序）：

```
#include <iostream>

using namespace std;

struct PlayerInfo
{
    int skill_level;
    string name;
};

using namespace std;

int main ()
{
    // 就像使用普通的变量类型一样，创建一个结构体的数组
    PlayerInfo players[ 5 ];
    for ( int i = 0; i < 5; i++ )
    {
```

```
        cout << "Please enter the name for player : " << i << '\n';
        // 首先使用正常的数组语法来访问数组元素
        // 然后使用“.”语法来访问结构体的域
        cin >> players[ i ].name;
        cout << "Please enter the skill level for " << players[ i ].name << '\n';
        cin >> players[ i ].skill_level;
    }
    for ( int i = 0; i < 5; ++i )
    {
        cout << players[ i ].name << " is at skill level " << players[ i ].skill_level
<< '\n';
    }
}
```

结构体PlayrInfo声明了两个域：玩家名称name和玩家技能等级skill_level。由于可以像使用其他变量类型（比如int）一样使用PlayerInfo，故你可以创建一个PlayerInfo的数组。创建了一个结构体数组后，你可以像访问简单类型数组中的元素那样，访问结构体数组中的每个元素：要访问数组中第一个结构体的某个域，比如获取数组中第一个玩家的名字，使用players[0].name即可。

此程序将数组和结构体结合起来，在第一个for循环里，读取了包括两块不同数据在内的五个不同玩家的信息，然后在第二个for循环中将这些信息显示出来。你不必再为每个玩家的数据都建立多个相关联的数组，也就是不需要单独建立player_names和player_skill_level数组。

11.1.2 传递结构体变量

你可能经常会想到写一个函数，将结构作为函数的参数或返回值。例如，如果你写了一个有移动飞船的小游戏，可能需要一个函数来初始化新出现的敌军的结构体：

```
struct EnemySpaceShip
{
    int x_coordinate;
    int y_coordinate;
    int weapon_power;
};

EnemySpaceShip getNewEnemy();
```

在此例中，调用getNewEnemy应该返回一个所有域都已初始化的结构体的值。你可以这样写：

```
EnemySpaceShip getNewEnemy()
{
    EnemySpaceShip ship;
    ship.x_coordinate = 0;
    ship.y_coordinate = 0;
    ship.weapon_power = 20;
    return Ship;
}
```

这个函数实际上是返回一个ship局部变量的副本。也就是说，它将会把该结构体的每个域都复制到新变量中。虽然复制多个域的过程看似缓慢，但没关系，因为大多数情况下计算机的速度快到可忽略不计这一复制过程的开销。然而，一旦你开始处理大量的结构体，问题便产生了！下一章我们将讨论如何使用指针来避免这些额外的复制操作。

为了获取返回的结构变量，可按如下方式书写代码：

```
EnemySpaceShip ship = getNewEnemy();
```

现在你能够像使用其他结构变量一样，轻松自如地使用ship变量了。

向函数传递一个结构体的代码应该像这样：

```
EnemySpaceShip upgradeWeapons (EnemySpaceShip ship)
{
    ship.weapon_power += 10;
    return ship;
}
```

将一个结构体传递到函数时，该结构体会被复制下来（就像刚才返回一个结构体那样）。我们在该函数中对结构体作出的任何修改都会丢失。这也就是在这个函数中对结构体进行修改后，将修改后的结构体作为函数的返回值返回的原因，因为原始的结构体[1]。

要使用upgradeWeapons来修改EnemySpaceShip，我们必须这样写：

```
ship = upgradeWeapons( ship );
```

当upgradeWeapons函数被调用时，变量ship被复制到函数的参数里；当upgradeWeapons函数返回时，返回的EnemySpaceShip变量被复制回ship中，覆盖了原来的域。

这是一个简单的程序，演示了如何创建和升级单个敌军飞船：

```
struct EnemySpaceShip
{
    int x_coordinate;
    int y_coordinate;
    int weapon_power;
};

EnemySpaceShip getNewEnemy ()
{
    EnemySpaceShip ship;
    ship.x_coordinate = 0;
    ship.y_coordinate = 0;
    ship.weapon_power = 20;
    return ship;
```

① 函数参数中的ship，这个参数的ship只是一个副本值，并不是原始值，并没有改变。——译者注

```
}

EnemySpaceShip upgradeWeapons (EnemySpaceShip ship)
{
    ship.weapon_power += 10;
    return ship;
}

int main ()
{
    EnemySpaceShip enemy = getNewEnemy();
    enemy = upgradeWeapons( enemy );
}
```

示例代码33：upgrade.cpp

你也许想知道，如何创建无限数量的敌舰，并在游戏过程中保持对所有敌舰的跟踪。怎么创建敌舰？你可以调用getNewEnemy函数。但怎么追踪这些敌舰呢？你会将它们存储在什么地方呢？目前，我们只能访问固定长度的数组。我们能创建一个EnemySpaceShip对象的数组：

```
EnemySpaceShip my_enemy_ships[ 10 ];
```

但这条语句一次最多给你10艘敌舰。这可能足够了，也可能不够。我们将在接下来的几章里介绍解决这个问题的方法。下一章将从指针谈起。

11.2 问答题

(1) 下列哪个选项访问了结构体b里的域？

A. `b->var;` B. `b.var;` C. `b-var;` D. `b>var;`

(2) 下列哪一项正确定义了一个结构体？

A. `struct {int a;}` B. `struct a_struct {int a};`
C. `struct a_struct int a;` D. `struct a_struct {int a;};`

(3) 下列的哪个选项正确地声明了类型名为foo，变量名为my_foo的结构体变量？

A. `my_foo as struct foo;` B. `foo my_foo;`
C. `my_foo;` D. `int my_foo;`

(4) 以下代码的最终输出结果是多少？

```
#include <iostream>

using namespace std;

struct MyStruct
```

```
{
    int x;
};

void updateStruct (MyStruct my_struct)
{
    my_struct.x = 10;
}

int main ()
{
    MyStruct my_struct;
    my_struct.x = 5;
    updateStruct( my_struct );
    cout << my_struct.x << '\n';
}
```

A. 5　　　　　B. 10　　　　　C. 此代码无法编译

11.3　实践题

(1) 编写一个程序，让用户可以在单个结构体中填入一个人的名字、地址，以及电话号码。

(2) 创建一个太空船对象的数组，并写一个程序，使太空船能够不断地更新位置，直到其在屏幕中无法显示为止。假设屏幕的尺寸是1024像素 × 768像素。

(3) 基于题(1)，创建一个地址簿程序。使用户不但能够填写单个结构体，还能够添加新条目，每一个新条目都要有自己的名称和电话号码。它允许用户添加任意多的条目；想一想，这做起来是否容易以及是否可行。该程序还应能够显示出全部或部分条目，使用户可以浏览条目列表。

(4) 编写一个程序，允许用户输入一个游戏的高分，并保持对用户名和分数的跟踪。添加一个功能，使之可以显示每个用户的最高分、特定用户的所有分数、所有用户的所有分数，以及用户的列表。

第 12 章

指针简介

12.1 忘记之前对指针的认知

很遗憾，指针的概念被很多初学者（甚至专业程序员）认为是一个神秘的东西。如果曾听说指针不易学习、让人困惑或是难以理解，请忘记和忽略一切关于这类“难”的说法。

事实上，在我以前教编程的时候，几乎每个学生都能很好地理解并掌握关于指针的知识。读完我的书，我保证你也能够理解指针的工作原理、作用及使用方法。当然，前提是你要花时间认真学习。

理解指针可能会花几天时间消耗些脑细胞，不过，给大脑简单做个“健身”运动也挺好的。我保证，接下来几章会被分成若干小块来写，这样你的大脑可以多多休息。在介绍语法细节前，我先解释指针的概念和作用。

12.2 指针的概念以及关注指针的原因

到目前为止，我们所能够使用的内存大小是固定的，这个大小在程序开始运行前就已经确定了。当你声明一个变量时，底层会分配出一定大小的内存来存储变量的信息，而分配内存的多少，则是在编译时确定的，在程序运行阶段，你不能改变分配的这块内存的大小。我们已经能够创建数据数组来使用大量的变量，其实质就是一大串内存。但是，数组不能够存储超过写程序时就已指定好的元素数量。在接下来的几章里，我们将学习如何访问比程序开始运行时所占用的更多的内存空间。你将学习如何创建数量无限制的敌舰，它们可以同时在四周飞行（当然，要减掉飞走的数量）。

为了能够访问（几乎）无限量的内存，我们需要一种类型的变量，它能够直接引用存储着变量值的内存，这种变量就叫做*指针*。

顾名思义，指针就是“指向”内存空间的变量。指针与超链接非常类似。一个网页存储于某个位置（即某个Web服务器）。你如果想将该网页的副本发送给某人，需要下载整个页面并通过

电子邮件发送给他吗？不，发送一个链接就好了。同样，一个指针允许你保存或发送一个到变量、数组或结构体的“链接”，而不是制作一份副本。

与超链接类似，指针存储着一些数据的位置，即*地址*。因此，你可以使用指针来保存从操作系统那儿获得的地址。换言之，使用指针使程序能够请求更多的内存，并且能访问这些内存。

实际上，你早就见过指针的例子了；当将一个数组传递给函数时，数组没有被复制，而是直接传递给这个函数了。这一过程就使用到了指针。看，指针没那么难理解！

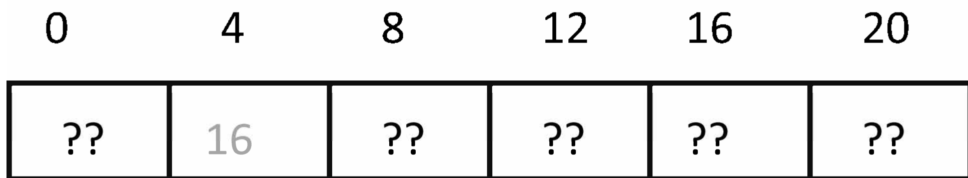
但在更深入地讨论之前，让我们先来谈谈内存。

12.3　内存的概念

有一个将内存概念形象化的简单方法，就是把它想象成一个Excel电子表格。电子表格由许多的“单元格”组成，每个单元格可以存储一段数据。计算机内存也类似：它由大量连续的数据段构成。而与Excel不同的是，内存里的每个“单元格”仅能存储一段非常小的数据——1字节，而1字节本身只有256个可能的取值（0~255）。而且，内存是一种“线性”的组织结构，而Excel是一种网格结构。事实上，你甚至可以将内存看做一个非常长的字符数组。

正如Excel里的每个单元格都可由行号和列号来定位一样，内存里的每个“单元格”也有一个地址。当指针保存了某个“单元格”的内存地址时，指针里存储的值即为该地址。（在Excel中，指针就是一个保存了另一个单元格的名称的单元格——比如，单元格C1里的内容是字符串A1。）

下面是一小块内存的示意图。它看起来很像一个数组，其实数组本身就是一串连续的内存。



此处的方框表示能够存储数据的内存空间，方框上方的数字是*内存地址*，每个数字都标识着一个内存位置。这些数字之间的步长为4，因为在内存中多数变量都占用4字节。此图表示与6个不同的4字节变量相关联的内存模型[①]。（顺便说一句，你经常会看到以十六进制格式表示的内存地址。如果你第一次见到这样的数字，可能会将它看做“无意义的数据”。不过，不用担心，我

① 其实，“一个变量在内存中占用4字节”这一说法，只有在32位机器上才成立（32位等于4字节）。多数32位CPU的指令都是以4字节为大小来操作数据的。即便如此，这种说法也不全对，因为还存在一些变量，其大小是大于4字节的（比如double类型的变量）。为了简化问题，现在还不需考虑这些细节。

会使用一般的数字表示方式[1]。）

地址4处的内存是一个指针变量，其存储的值可以是另一个变量的内存地址——16。其他标记为??的值表明它们不具有任何特定的已知值；当然，任何时刻每个内存地址里都存储着某些值。在该内存块被初始化之前，这个值是没有用的，可以是任何值。

12.3.1 变量与地址

你可能会很困惑，变量和地址究竟有什么区别？变量是值的表示形式，这个值实际上存储在一个特定的内存位置，即一个特定的内存地址。换句话说，编译器使用内存地址来实现程序中的变量。指针是一种特殊类型的变量，可以存储一个变量的地址。

最酷的是，一旦你拥有了变量的地址，就可以从这个地址中取出存储在其中的数据。如果你恰好要将一个巨大的数据块传递到一个函数中，在程序运行时刻将数据块的位置传递给函数，要比复制数据块的所有数据高效得多（类似于刚才针对数组的操作）。这个方法同样可以用来避免传递结构体的副本到函数。我们的思路是将该结构体中数据的内存地址传递给函数，而不是将该结构体的数据复制一份。

指针最重要的功能是让你在任何时候都可以从操作系统里获取更多的内存。怎样从操作系统里获取内存呢？[2]操作系统会告诉你该内存的地址，你需要用一个指针将其存储下来。如果你之后又需要更多内存的话，可以向操作系统申请更多的内存，并更新指针的值。因此，指针使我们可以使用超过固定大小的数据，在程序运行时动态选择需要的内存大小。

1. 相关术语的说明

指针可以指：

(1) 内存地址本身；
(2) 存储内存地址的变量。

通常，两者之间的区别并不重要。因为如果你传递一个指针变量给函数，就是在传递该指针的值，也就是内存地址。

当我想讨论一个内存地址时，会称它为内存地址，或仅仅称为地址；当想讨论一个存储着内存地址的变量时，我才会称它为指针。

---

① 十六进制数以16为基底，通常以这样的格式来书写：0x10ab0200，0x表示这是一个十六进制数，剩余的是数字，其中A~F表示数值10~15。

② 操作系统管理内存，这一说法并不全对。事实上，通常会有几个不同“层”的代码来处理内存分配。操作系统是其中一层，但在操作系统上面还有其他层。为了避免混乱，我们先暂且忽略这些区别。如果你不能完全理解，请不要担心。如果它很重要的话，我是不会做个注释就草草了事的。现在姑且不必理会它，稍后它才会有意义。

当一个变量存储了另一变量的地址，我会说，它指向了那个变量。

12.3.2　内存布局

内存究竟从何而来？为什么请求内存时无论如何都要通过操作系统呢？

Excel中有非常多的单元格供你使用。计算机中也有非常多的内存供你访问。但是，内存比Excel更结构化。你的程序可使用的内存中，有一些已经在使用。当前正在执行的函数中声明的变量就正存储于内存中，这部分在使用的内存称为*栈*。之所以称为栈，是因为如果你调用了若干个函数，那么系统会按照函数的调用顺序，将每个函数的局部变量以“栈叠加”的方式放置于这段内存中。我们目前使用到的所有变量都存储在栈上。

内存的第二部分是*未分配内存区域*（free store），有时又称为*堆*。这是一片尚未分配的内存区域，你可以以块为单位来请求它。这部分内存由操作系统进行管理，一旦一块内存被分配出去，它便只能由分配了这块内存的原始代码使用，或是由内存分配器将这块内存的地址交付给代码来使用。使用指针，我们便可以访问这块内存。

能够访问内存是很强大的。但能力越大，责任也越大。内存是稀缺资源，虽然不像在GB级的RAM成为标准之前那么稀缺，但它仍然是有限的。每一块从未分配内存区域中分配出来的内存，当你的程序不再需要它时，都应释放回去。负责释放特定内存块的代码称为该内存的*所有者*。当内存的所有者不再需要该内存时，例如在一个空间射击游戏中，如果一艘船被摧毁，这艘船的内存的所有者就应该将该内存释放回未分配内存区域，以便这块内存可以重新分配给其他代码语句使用。如果不这样做，程序将会耗光内存，导致运行速度下降甚至崩溃。你或者你身边的人可能抱怨过火狐浏览器（Firefox）占用了太多的内存，导致浏览器越来越慢，像龟爬似的，这就是因为某些开发者没有释放本应释放的内存，造成了所谓的*内存泄漏*[①]。

所有权这个概念是函数及其使用者之间接口的一部分，它在编程语言中没有显式出现。当你写一个函数，它接受一个指针，你应该说明该函数是否占用了内存的所有权。C++不会为你追踪内存的所有权。只要程序正在运行，C++就永远不会帮你释放已经显式分配了的内存，除非你显式要求释放。

事实上，只有某些代码应使用某些内存，这就是为什么不能随便取一些内存地址来使用；如果只是生成一个随机数，然后把它当做内存地址来用，后果会怎样呢？技术上来讲可以这样做，但这是一个糟糕的想法。你不知道谁被分配了这块内存，甚至有可能是栈本身，如果你修改了内存里存储的值，就会破坏正在使用中的数据！为了帮助发现这类问题，操作系统会将尚未分配给你使用的内存保护起来——该内存对你来说是*非法的*，访问非法内存将导致程序崩溃，这样，你

① 我得为火狐辩解下。有些问题可能是由于写得很差的扩展（即用户写的附加组件）造成的，不能怪罪于火狐内核。不过，最终的结果是一样的：运行时的内存不足给用户带来了严重的后果！

就能察觉到这类问题。[①]

等一下，我说崩溃也是好事一桩吗？嗯，的确是！相比将错误数值写入合法内存中引起的错误，访问非法内存造成的崩溃更容易被发现。你通常会很快发现此类崩溃，因为问题立即就会发生。如果改变了本不属于你的内存，这个错误会持续到拥有该内存的代码尝试去使用它时才会发生，而这可能距内存被修改已经很长时间了。我的一个同事喜欢把它解释为："轮胎脱落时，车轮螺母一公里前就掉了。"祝你找螺母好运！

顺便说一句，有些人会告诉你，非法内存造成的崩溃很难诊断，这是因为那些人没读过这本书。第20章将讨论如何调试由非法内存造成的崩溃。

1. 非法指针

一种可能不小心访问非法内存的情况是：使用了未初始化过的指针。声明一个指针时，指针中的数据是随机生成的，它指向一个可能合法，也可能非法的位置。不过可以肯定的是，此时使用它相当危险，这等同于使用了一个随机生成的地址！使用此数值可能导致程序崩溃，或数据损坏。你必须在使用指针前初始化它！

2. 内存和数组

还记得我说过么？越过数组末端写数据会发生问题，对吧？现在，我们知道了更多关于内存的知识，你可以理解为什么了吧。数组具有一段与其关联的特定数量的内存，数量的多少由数组的大小决定。如果你访问数组末端之后的元素，访问的就是与数组不相关的内存；没错，这块内存不在数组中。而这块内存究竟是什么，这取决于实际的代码和编译器的实现方式。但它不会是数组的一部分，所以使用它肯定会产生问题。

12.4　指针的其他优点（和缺点）

现在，你已经了解了一些指针的细节。我们回想一下之前的比喻，再来权衡下一些使用指针的利弊。超链接和指针有很多相同的优点和缺点。

(1) 不必做复制；如果该网页又大又复杂，复制可能很难（比如将整个维基百科的一份副本发送给某人）。同样地，内存中的数据可能相当复杂，它可能很难被正确地复制（后面将详细讨论），或者复制的速度太慢（复制大量的内存可能会非常耗时）。

(2) 不必担心获得的是否是网页的最新版本。如果作者更新了网页，只要重新访问该链接就能得到更改的内容。如果你有一个指向内存的指针，就总是能够访问那个内存地址的最新值。

① 随机生成的内存地址还存在一个小问题，即内存地址一般需要对齐。要访问一个整型数，所使用到的内存地址必须是4的倍数。如果你随机生成一个内存地址，必须正确对齐。不同的计算机体系结构对内存对齐有着不同的要求，由于对性能的需求不同，这种现象普遍存在。

当然，发送链接而不是副本，也存在一些不足之处。

(1) 页面可能被移动或删除。类似地，即使指针仍指向某块内存，该内存还是有可能已经被释放回了操作系统。为了避免这类问题，拥有这块内存的代码必须跟踪并确定是否有其他人正在使用它。

(2) 你必须在线才能访问页面。这是超链接的缺点，但通常不会影响指针。

将指针比喻为Web上的链接，可以帮助你理解为什么要使用指针，但也有些问题。一是超链接和Web是不同的东西，而指针和变量则不是。这是什么意思呢？指针只是另一种变量（但它有其特殊性质），而超链接不是网页。但从另一方面来说，指针不同于普通类型的变量，就像是超链接不同于网页一样。

到目前为止，全清楚了吗？我答应过将指针的内容分成很多的短章，以便你的大脑能休息一下。所以，本章就先到此为止吧。现在你已经掌握了一些所需的核心知识，下一章将讨论使用指针的具体细节。

12.5　问答题

(1) 以下哪项不是使用指针的好理由？

A. 你想要允许函数修改传递给它的参数
B. 你想要避免复制一个占用了很大内存的变量，以节省空间
C. 你希望能够从操作系统获得更多的内存
D. 你希望能够更快速地访问变量

(2) 指针中存储的是什么？

A. 另一个变量的名称
B. 一个整数值
C. 另一个变量在内存中的地址
D. 一个内存地址，但不一定是另一个变量

(3) 程序执行过程中，从哪里可以获取到更多的内存？

A. 不能得到任何更多的内存　　B. 栈
C. 未分配内存区域　　D. 通过声明另一个变量

(4) 使用指针时，可能会遇到什么错误？

A. 访问了本不能用到的内存，导致崩溃
B. 访问错误的内存地址，导致数据污染

C. 忘了将内存释放回操作系统，导致程序耗光内存
D. 以上皆可能

(5) 函数中声明的普通变量，其内存来自哪里？

A. 未分配存储区域
B. 栈
C. 普通变量不使用内存
D. 来自该程序的二进制文件本身（这就是为什么EXE文件会这么大！）

(6) 分配到内存后，需要做些什么？

A. 什么都不用做，它永远是你的
B. 使用完后要释放回操作系统
C. 将所指向的值置为`0`
D. 将值`0`存入指针中

12.6　实践题

(1) 找一个你之前写的小程序，比如本书前面几章的一道实践题。找出所有的变量，想象一下每个变量都具有与之关联的内存。试着画一个盒子图，就像前面我用来表示每个变量及与之关联的内存的图。怎样能将一串不属于同一个数组的变量表示出来？注意，即使不属于同一数组，这些变量在内存中也是一个接一个连续排放的。

(2) 思考以下程序需要多少内存：

```
int main ()
{
    int i;
    int votes[ 10 ];
}
```

变量`votes[0]`、`votes[9]`和`i`在内存中的位置，你能够确定的有哪些？（提示：你可能不确定`i`的内存地址，但能知道它一定不在哪里。）尝试画出这个程序可能的内存布局。

第 13 章 使用指针

现在，我们已经了解了内存的概念，以及应如何理解内存。那么，如何编写使用内存的代码呢？在本章中，你将学会使用指针的语法，通过大量图表和一些基本例子来学习真正的程序是如何使用指针的。下一章才会介绍如何完全访问未分配内存区中的内存，而本章你只需掌握做到这一点的所有工具。

13.1 指针的语法

声明一个指针

C++有专门的语法来声明一个指针变量。该语法不仅仅指出一个变量为指针，同时表明指针所指向的内存的类型。

以下是声明一个指针变量的语法：

```
<type> *<ptr_name>;
```

例如，可以声明一个指针，用它存储一个整型数的地址：

```
int *p_points_to_integer;
```

注意，这里的*是关键所在，它紧挨着出现在变量名前时，表示声明该变量为指针。我习惯在变量名前加个p_前缀，以便清楚地表明该变量是个指针，但这不是C++的语法所必需的。有个小陷阱：要在一条声明语句中声明多个指针，就必须在每个变量名前都加上星号：

```
// p_pointer1是指针，nonpointer1是普通的int变量
int *p_pointer1, nonpointer1;

// p_pointer1和p_pointer2都是指针
int *p_pointer1, *p_pointer2;
```

你可能会奇怪：为什么没有一种更简单的方式来声明指针呢？比如用`pointer p_pointer`这样的语句。这是因为，要让编译器能够正确地解释和使用内存地址，就需要让它知道地址中存

储的是哪种类型的数据。（比如，内存中相同的字节数对于double和int类型来说，意义却不一样。）与其为每种指针类型创建单独的名字（比如用int_ptr表示int类型的指针，char_ptr表示char类型的指针，等等），还不如总是用*和类型名来声明指针。

13.2 指针的指向：变量的地址

指针既可以直接指向新分配的内存，也可以指向一个已经存在的变量。来看看这要怎么做。为了获得变量地址（即变量在内存中的位置），要把符号&放在变量名前。&称为取地址操作符，因为它能返回变量的内存地址：

```
int x;
int *p_x = & x;
*p_x = 2; // initialize x to 2
```

&的作用是得到变量的地址，有一种有效的方式可以记住它：字符“&”的单词（ampersand）和“地址”（address-of）都是以“a”开头的。使用&符号就像是通过网站的地址栏获得该网站的URL[①]，否则，我们只能光盯着网页的内容看了。

获取变量地址的意义通常是为了做些独特的事情——大多数时候，是想从变量中获得其实际的值。

指针的使用

使用指针同样也需要一些新的语法。指针的使用通常可以用来做下列两件事：

(1) 获得指针中存储的内存地址；
(2) 获得内存单元中存储的值。

要获得指针中存储的内存地址，直接使用指针即可，就像使用一个普通变量一样。

下面的程序片段输出指针p_pointer_to_integer指向（存储）的地址：

```
int x = 5;
int *p_pointer_to_integer = & x;
cout << p_pointer_to_integer; // 输出x的地址
// 等价于cout << & x;
```

这个代码片段打印输出变量x的内存地址，而这个变量存储在p_pointer_to_integer中。

要访问内存单元中存储的值，你可以使用*操作符。下面就是一个小例子，它初始化一个指向另一变量的指针：

① URL，统一资源定位符，即我们常说的网址。——译者注

```
int x = 5;
int *p_pointer_to_integer = & x;
cout << *p_pointer_to_integer; // 输出5
// 等价于cout << x;
```

代码*p_pointer_to_integer表示“到指针所指向的内存，去取出存储在里面的值”。此例中，指针p_pointer_to_integer指向了变量x，而x的值是5，所以输出了数值5。

有个简单的方法，可以记住*用于获取指针变量所指向的变量值：指针变量跟普通变量没什么两样儿，我们可以通过变量名来获得变量的值，而指针变量的值就是它存储的内存地址。如果我们还想做一些更复杂的事儿，比如获得内存地址中存储的值，就必须使用特殊的语法，即使用*号。*被看做特殊行为的标志，就像有人会在巴里[①]家旁边放一个星号，来表示这家主人是打棒球的。

使用*来获得指针变量指向的地址的值，这一过程称为*间接引用指针*。这个名词的由来是为了获得地址中存储的值，我们是通过一个到该内存地址的引用，使用它，间接地达到目的地。

通过间接引用指针，还可以修改指针地址所指向的变量的值：

```
int x;
int *p_pointer_to_integer = & x;
*p_pointer_to_integer = 5;  // x 现在为 5!
cout << x;
```

那么，什么时候应该在变量名前加上*号（或&号）呢？这个其实很容易出错。下表可供参考：

操作目的	需要的操作符	示　　例
声明指针	*	int *p_x;
获得指针所指向的地址	不需要	cout << p_x;
调整指针所指向的地址	不需要	int *p_x; p_x = /*address*/;
获得指针所指向的地址中的值	*	cout << *p_x;
调整指针所指向的地址中的值	*	*p_x = 5;
声明变量	不需要	int y;
获得变量的值	不需要	int y; cout << y;
调整变量的值	不需要	int y; y = 5;
获得变量的地址	&	int y; int *p_x; p_x = & y;
调整变量的地址	不可行	不可以。变量地址不能更改

① 巴里，美国棒球巨人。——译者注

记住这张表，只需要两个简单的规则：

指针存储的是地址。因此，直接使用“裸”指针[①]得到的就是地址。要获得或调整存储在该地址中的值，必须添加额外的*。

变量存储的是数据值。因此，直接使用变量得到的就是数据值。而要获得变量的地址，就必须额外添加&。

现在，我们通过一个简单的程序，来演示这些功能，并学习一个很实用的分析内存变化情况的技巧。

```
#include <iostream>

using namespace std;

int main ()
{
    int x;              // x为普通变量
    int *p_int;     // p_int为指向一个整型数的指针

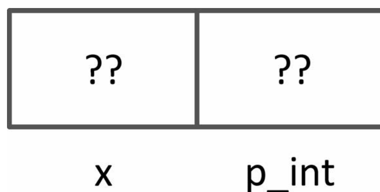
    p_int = & x;    // 将x的地址赋值给p_int
    cout << "Please enter a number: ";
    cin >> x;        // 读入一个值并赋值给变量x，这里的x也可以用*p_int来代替
    cout << *p_int << '\n'; // 使用*来获得指针所指向的变量的值
    *p_int = 10;
    cout << x;  // 再次输出10!
}
```

示例代码34：pointer.cpp

第一个cout输出变量x的值。这是怎么发生的呢？让我们逐步地执行程序，观察内存是怎样变化的。我们用箭头来表示指针指向的位置，方框中的数字表示非指针变量在内存中的值。

刚开始，我们有一个整型变量x，以及一个指向整型的指针变量p_int。

直观上，可以认为现在有两个值未知的变量（它们可能彼此相邻）。

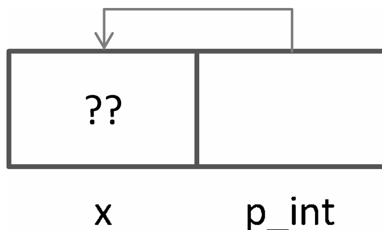


接着，代码通过使用取地址操作符（&）获得变量x的地址，并将该地址存储到指针p_int中。

```
p_int = & x;            // 将x的地址赋值给p_int
```

① “裸”指针，即不带任何符号的指针。——译者注

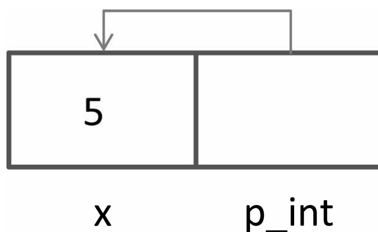
因此，我们可以画一条从变量p_int到变量x的箭头，表示指针p_int指向变量x。



然后用户输入一个数字，存储在变量x中，该存储位置也是p_int所指向的位置。

```
cin >> x;          // 读入一个值并赋值给变量x，这里的x也可以用*p_int来代替
```

简单起见，我们假设用户输入数字5。现在内存的情况变成了这样：



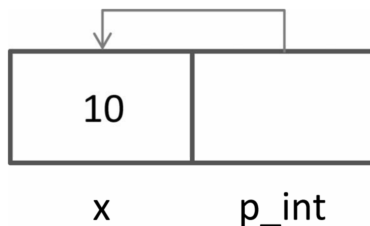
接着，下一行代码将*p_int传给cout。*p_int间接引用了p_int，它会检查p_int中的地址，并且到该地址中取出其变量值，你可以结合内存示意图中的箭头来想象这一过程。

```
cout << *p_int << '\n'; // 使用*来获得指针所指向的变量的值
```

最后的两行语句表明，通过指针可以修改变量原来的值。这个语句将值10存储到p_int所指向的内存中，也即是存储着变量x的值的内存。

```
*p_int = 10;
```

现在的内存状态是：



你看，通过示意图，我们可以很容易弄明白使用指针时内存的变化过程。当你搞不清楚时，绘制出内存的初始状态，配合箭头图逐步运行程序，内存的变化过程就一目了然了。每当指针的

指向改变时，便绘制新的箭头；每当变量的值发生变化时，更新它的值。通过这些操作，即使再复杂的系统，你也能够理解。

13.3 未初始化指针与空指针

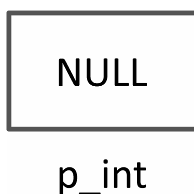
注意到上面的例子中，指针p_int在使用前先被初始化指向了一块特定的内存地址。如果不这样做，指针可能会指向任何位置，从而导致令人不快的后果，比如覆盖了其他变量的值，或程序崩溃，等等。为了避免此类事故或其他不良后果，你应该养成在使用指针前先初始化的好习惯。

有时候，你需要明确知道："哟，这个指针一看就没有被初始化。"因此，可以用NULL这个C++的特殊值，来标记一个明确没有被初始化的指针。如果一个指针指向NULL（即指针存储的值为NULL），即说明它未初始化。每新建一个指针，你应该首先将它的值设置为NULL，这样可以方便以后检查，看看它是否已经被设置成了指向可用的地址。否则，就没有办法测试指针是否可用，这可能会导致系统崩溃。

```
int *p_int = NULL;

// 可能设置，也可能不设置p_int的代码
if ( p_int != NULL )
{
    *p_int = 2;
}
```

在内存示意图里，要表示一个NULL指针，可以简单地将指针的值写成NULL，而不必画一条指向NULL的箭头。



13.4 指针和函数

指针允许你将局部变量的地址传给函数，然后在函数中修改局部变量。下面两个函数是说明这一过程的经典例子。这两个函数都试图交换两个变量中存储的值：

```
#include <iostream>

using namespace std;

void swap1 (int left, int right)
```

```
{
    int temp;
    temp = left;
    left = right;
    right = temp;
}

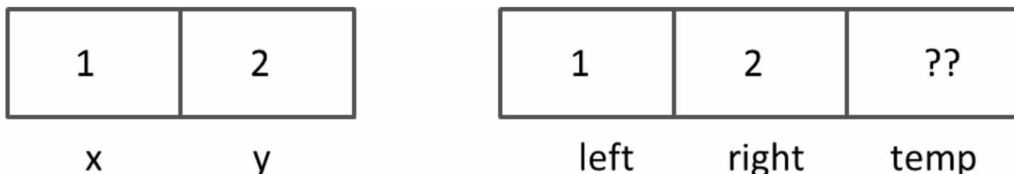
void swap2 (int *p_left, int *p_right)
{
    int temp = *p_left;
    *p_left = *p_right;
    *p_right = temp;
}

int main ()
{
    int x = 1, y = 2;
    swap1( x, y );
    cout << x << " " << y << '\n';
    swap2( & x, & y );
    cout << x << " " << y << '\n';
}
```

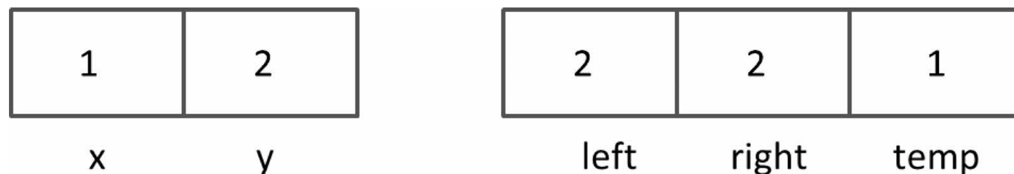
示例代码35：swap.cpp

先思考下，你猜哪个函数能正确地交换了两个值呢？

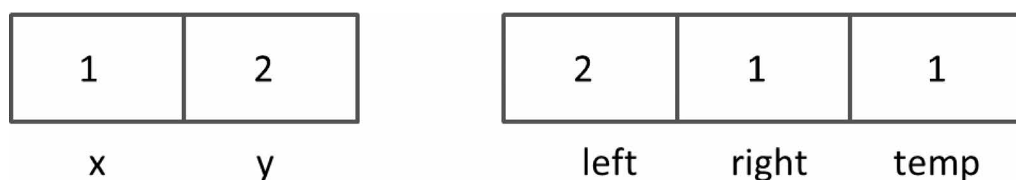
没错，swap1只是交换了swap1函数中两个局部变量的值，而无法修改传递过来的原始值，因为swap1只是存储了原始值的一个副本（原始值在变量x和y中）。直观上看，swap1函数调用和复制了变量x和y的值，并传给变量left和right：



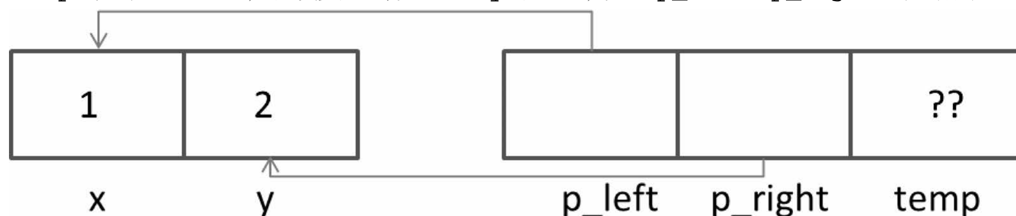
接着，left的值被放入temp，而left被赋给right的值：



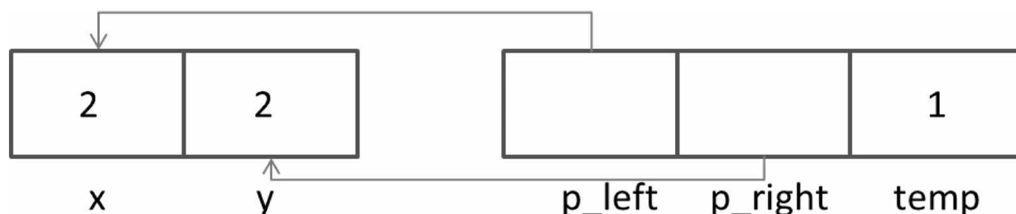
最后，temp的值被放入right。即，交换了left和right的值，但x和y的值完全没变：



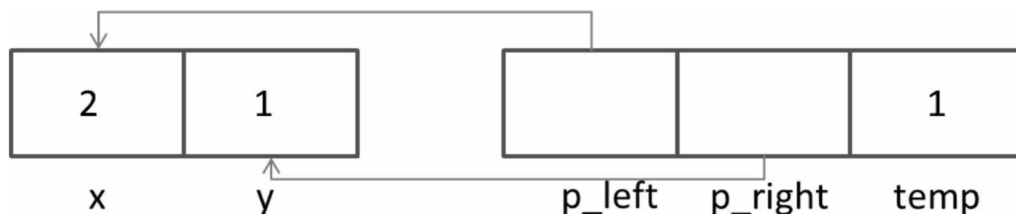
函数swap2就有意思多了，它接受局部变量x和y的地址，变量p_left和p_right现在指向了x和y：



现在，swap2可以访问变量x和y的内存。所以，当swap2执行交换操作时，写入的是这两个变量的内存。首先，将p_left指向的值复制给变量temp，然后，又将p_right所指向的值复制给p_left：



请注意，此刻变量x在内存中的值已经被修改了。最后，temp中的值赋给p_right所指向的内存，完成了交换：



像这样去交换两个变量的值，并不是指针的主要价值。C++的另一种语言特性，可以很容易地写出这类交换函数，无需通过完整的指针。这一语言特性即“引用”。

13.5　引用

有时，你想利用指针的一些特性（如避免大块数据的额外副本），但不又需要其全部功能。这种情况，往往可以使用引用。引用是指一个变量引用了另外一个变量，它们背后共享着相同的内存。引用变量的使用方法跟普通变量类似，你可以将它想象成一个简化版的指针，我们不需要在使用引用的值或给引用赋值时，使用特殊的*和&语法。与指针不同的是，引用必须始终指向有效内存。声明一个引用，需要使用&符号：

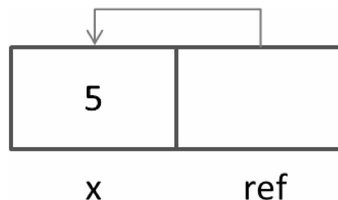
```
int &ref;
```

然而，这个声明是非法的，因为引用必须初始化（引用必须总指向有效的地址）。

```
int x = 5;
int &ref = x; // 注意，不需要在x的前面加上*号!
```

我们可以用跟可视化指针相同的方式来可视化引用。但是，使用引用时，得到的是引用的内存中的值，而不是内存地址。

```
int x = 5;
int &ref = x;
```



这里，变量ref的实际地址持有一个指向变量x的内存的指针。当你直接写ref（即不带任何诸如&或*符号）时，编译器知道你想要指向的实际的值。从某种意义上说，引用是一种跟指针的“默认”行为相反的指针，当你使用变量名时，结果刚好跟使用指针相反。

给函数传递结构体时可以使用引用，而无需传递整个结构体，也不用担心空指针问题。

```
struct myBigStruct
{
    int x[ 100 ]; // 占用了大量内存的大结构体!
};

void takeStruct (myBigStruct& my_struct)
{
    my_struct.x[ 0 ] ="foo";
}
```

由于引用始终指向初始对象，所以你可以避免复制整个对象，并能够修改传递到函数的原始

对象。上面的例子演示了如何修改my_struct.x[0]，使得函数返回时，传递到函数中的原始结构体包含foo。

我们刚才学习了使用指针来写一个交换函数的方法，使用引用来写会更简便。现在来看一看：

```
void swap (int& left, int& right)
{
    int temp = right;
    right = left;
    left = temp;
}
```

注意，这远比使用等效的指针更简单。事实上，我们可以把引用仅仅看做原始变量的一个别名。当然，在编译器中引用的实现使用了指针来存储，只不过获取真正的数据和间接引用等工作编译器都帮你做了。

引用与指针的区别

当需要通过多个名称来使用同一个变量时，我们可以使用引用来替代指针。比如，你想要将参数传递给一个函数而不用复制它们，又或者希望函数对参数的修改能够对调用者可见。

引用并不像指针那般灵活，因为引用必须总是有效的。不存在空引用（null reference）——我们不能在使用引用时说“嘿，我引用的东西无效”，因为这不是设计引用的目的。由于引用不能指向NULL，所以不能用它来构建复杂的数据结构。接下来的几章，我们会更多地讨论数据结构的构建。每次构建数据结构前，先问问自己，你是否可以用引用来达到同样的效果。

引用和指针还有一个区别：一旦一个引用被初始化，你便不能改变它指向的内存。引用永远指向相同的变量，这也限制了它们构建复杂的数据结构的灵活性。

剩下的几章中，我会在适当时候使用引用。尤其是将一个结构体或者类（我们以后会讲到类）的实例作为参数传给函数时，总会使用引用。使用引用的模式通常会是这样：

```
void (myStructType& arg);
```


13.6　问答题

(1) 下面哪个选项正确地声明了指针？

A. int x;　B. int &x;　C. ptr x;　D. int *x;

(2) 下面哪一项给出了整数变量的内存地址？

A. *a;　B. a;　C. &a;　D. address(a);

(3) 下面哪一项给出了指针p_a所指向的变量的内存地址?

A. `p_a;` B. `*p_a;` C. `&p_a;` D. `address( p_a );`

(4) 下面哪一项给出了指针p_a所指向的地址中存储的值?

A. `p_a;` B. `val( p_a );` C. `*p_a;` D. `&p_a;`

(5) 下列哪一项正确声明了一个引用?

A. `int *p_int;` B. `int &my_ref;`
C. `int &my_ref = & my_orig_val;` D. `int &my_ref = my_orig_val;`

(6) 下列哪一项不适合使用引用?

A. 为了存储一个未分配存储区中动态分配出来的地址
B. 为了避免一个较大的值传递给函数时的复制操作
C. 为了强制函数的一个参数，其值永远不能为NULL
D. 为了让一个函数能够访问传递给它的原始变量，而无需使用指针

13.7 实践题

(1) 写一个函数，提示用户输入姓氏和名字，姓和名要作为两个单独的值。这个函数通过传递到该函数的额外的指针（或引用）参数，返回两个值给调用方。先试着用指针来做，然后尝试使用引用。（提示：函数签名看起来类似于前面的交换函数！）

(2) 绘制出题(1)中所写的函数的示意图，参考我画的交换函数的示意图。

(3) 修改实践题(1)中所写的程序，使得它不必总是提示用户输入姓氏。只有调用方传递进来的姓氏为空指针时，才提示。

(4) 写一个函数，它接受两个输入参数，返回两个结果给调用方：一个是两个参数相乘的结果，另一个是相加的结果。由于函数只能直接返回一个值，你需要通过指针或引用参数来返回第二个值。

(5) 写一个程序，比较两个不同的变量在栈中的内存地址，并根据地址的数值顺序依次输出变量。这个顺序有没有让你大吃一惊?

第 14 章

动态内存分配

14

恭喜你熬过了前面几章枯燥的内容，现在一起进入指针的有趣部分：使用指针来解决实际问题。没错！我们终于为学习如何在程序运行时获得尽可能多的内存做好了准备。虽然不太应该，但我们已经有能力“垄断”内存市场了！

14.1 获得更多的新内存

*动态分配*是指，在程序运行时请求所需要的内存大小。你的程序将计算出它所需的内存数量，而不是只能处理一组特定大小的固定的变量。本节将讲述如何分配内存的基础知识，后续各节将介绍如何充分利用动态分配的优势。

首先让我们来看看如何获得更多的内存。关键字new用来将未分配内存区中的内存初始化分配给指针。请记住，未分配内存区是一块未使用的内存，你的程序可以请求访问它。以下是基本的语法：

```
int *p_int = new int;
```

new运算符需要一个“示例”变量，以便计算出所请求的内存大小。在这个例子中，示例变量是一个整数类型（int）。因此，new运算符接受一个整数类型，并返回足够的内存来容纳整数值。

p_int设置为指向该内存，使得p_int和使用p_int的相关代码成为了该内存的所有者。换句话说，使用p_int的代码必须在不再使用该内存时，进行一个称为*释放内存*的操作，显式地将这块内存归还给未分配内存区。在p_int释放之前，它所指向的内容被标记成了“正使用”（in-use），不能再次分配。如果你一直分配内存但又不释放它，内存将耗光。

要“释放内存”，可以使用delete关键字，它把通过new分配到的内存释放回未分配内存区。以下语句释放p_int指向的内存：

```
delete p_int;
```

释放指针指向的内存后，将指针重置为指向NULL是个不错的选择：

```
delete p_int;
p_int = NULL;
```

这不是必要的。不过，指针一旦被delete，你就不能读写它指向的内存了，因为这块内存已经释放回未分配内存区（而且很可能又分配出去）。将指针设置为NULL后，如果代码尝试间接引用一个释放了的指针（即使是经验丰富的程序员也常常犯这种错误），你立即就能发现，因为该程序会崩溃。这种情况比用户数据被破坏后才发现要好得多。

14.1.1　运行内存不足

内存不是无穷无尽的资源。我们的确可以任意挥霍内存，但如果真这么做了，将无法获得更多的内存。在C++中，如果因为系统内存不足而导致调用new失败，系统将“抛出一个异常”。通常不用担心这些，现代操作系统中这种情况极其罕见，许多程序可以忽略掉这种可能性。（如果程序写得很好并正确释放内存，它就更不可能发生了。一个永远不释放内存的程序，才最有可能引起内存不足。）异常是接近本书结尾才介绍的高级内容。通常最好的做法是：始终释放你分配到的内存，别担心new操作会失败。

14.1.2　引用和动态分配

一般来说，不宜将刚分配到的内存存储于引用中：

```
int &val = *(new int);
```

其原因是：引用不直接访问原始的内存地址。虽然可以通过&来访问到，但是引用一般用在为变量提供额外的名称，而不是存储动态分配的内存。

14.2　指针和数组

你可能有疑问，现在已经可以用new来将指针初始化指向一块内存了，那如果想实际获取更多内存应该怎么办呢？答案是，指针也可以指向一组值的序列。换句话说，指针可以像数组那样使用——毕竟，数组就是顺序布局在内存中的一组值的序列。由于一个指针存储一个内存地址，所以它能够存储数组的第一个元素的地址。要访问数组的各个元素，你只需知道该元素与数组起始地址的距离，而这个距离是固定的。

这有什么用呢？实际上，我们可以从未分配内存区中动态地创建一个数组，在运行时确定需要的内存数量。稍后我会展示一个这样的例子，现在先来了解一些基础知识。

可以像这样，将一个数组直接赋值给指针，无需用到取地址操作符：

```
int numbers[ 8 ];
int* p_numbers = numbers;
```

现在，可以像使用数组一样地使用指针：

```
for ( int i = 0; i < 8; ++i )
{
    p_numbers[ i ] = i;
}
```

数组numbers被赋给指针时，仿佛它本身就是一个指针一样。重要的是理解清楚，数组不是指针，但数组可以被赋值给指针。C++编译器知道怎样将一个数组转换为一个指针，这个指针会指向数组的第一个元素。（这种转换在C++中经常发生。例如，你可以将一个char类型的变量赋给一个int类型的变量。char不是int，但编译器知道如何进行转换。）

可以使用new动态来分配一个数组的内存，并将该内存赋给指针：

```
int *p_numbers = new int[ 8 ];
```

这条语句使用了数组的语法来作为new的参数，以便告诉编译器需要分配多少内存：只要8个元素的整数数组就够了。现在，你可以像使用数组一样地使用指针p_numbers。不过与数组不同的是：需要释放p_numbers指向的内存，但你从来不需要释放一个指向静态声明的数组的指针。有一个delete运算符的特殊语法，可以释放动态分配的数组内存：

```
delete[] p_numbers;
```

方括号告诉编译器，指针指向了一个数组，而不是单个值。

现在，到了你一直在翘首企盼的例子了：动态地确定需要多少内存：

```
int count_of_numbers;
cin >> count_of_numbers;
int *p_numbers = new int[ count_of_numbers ];
```

这段代码会询问用户需要的内存数量，然后使用该变量来确定动态分配的数组的大小。事实上，我们甚至不需要预先知道确切的数字，而是随着数量的增长重新分配内存。这意味着可能要做一些额外的复制操作。接下来看一段演示这种方法的程序，该程序读入用户输入的数字。一旦数字的数量超过了数组所能容纳的大小，我们就会重新调整数组。

```
#include <iostream>

using namespace std;

int *growArray (int* p_values, int cur_size);

int main ()
{
    int next_element = 0;
    int size = 10;
    int *p_values = new int[ size ];
    int val;
    cout << "Please enter a number: ";
```

```
    cin >> val;
    while ( val > 0 )
    {
        if ( size == next_element + 1 )
        {
            // 现在需要实现growArray
            p_values = growArray( p_values, size );
        }
        p_values[ next_element ] = val;
      cout << "Please enter a number (or 0 to exit): ";
      cin >> val;
    }
}
```

示例代码36：resize_array.cpp

来考虑一下如何增长数组。怎么做呢？我们不能只是请求下扩展内存就万事大吉了。这与Excel不一样，我们不能在想要更多内存空间时直接添加一个新列。我们必须重新请求更多的内存，并把原来的值复制过来。

另一个问题是应该请求多少内存。一次增长一个数组元素的空间实在有些低效：尽管你不会耗尽内存，但这会导致许多次不必要的内存分配操作，使得速度太慢。一个好的策略是把当前的数组大小加倍。这样一来，如果停止读入新值，这不会浪费太多空间——总共占用的空间不会超过使用中的空间的两倍，同时又不必不停地重新分配内存。显然，我们需要知道当前数组的大小以及原始数组的值，以便复制原始数组。

```
int *growArray (int* p_values, int *size)
{
    int *p_new_values = new int[cur_size*2];
    for ( int i = 0; i < cur_size; ++i )
    {
        p_new_values[ i ] = p_values[ i ];
    }
    delete p_values;
    return p_new_values;
}
```

示例代码37：resize_array.cpp（待续）

请注意这段代码是如何在数组数据复制完成后，小心翼翼地删除掉p_values的值的。一不留神，就会发生内存泄漏，因为程序从growArray返回后我们覆盖了指向原数组的指针。

14.3　多维数组

调整一个大数组的大小是非常有用的技术，你一定要掌握。但是，有时你想要处理的不仅仅是一个大数组。还记得当我们讨论多维数组时，心情何等激动吗？如果能够选择多维数组的大小，是不是很厉害呢？我们完全可以做到这一点。这是一个很好的帮助你真正深入理解指针的练习，而且它本身也非常有用。不过，做到这一点需要掌握一些其他的背景知识。本章接下来的几个部

分将介绍这些内容，最后再向你展示如何动态分配多维的数据结构。

14.4　指针运算

> 本节将深入探索指针，可能需要多花点工夫来学习。但这些极富挑战性的内容很有意义，如果你一遍不能理解清楚，那最好再读一遍。如果能理解本节的所有内容，包括二维数组的分配，那么有关指针的一切你几乎都能够不太费力地掌握了。所以，本节有点艰难，而且不像某些章那样立竿见影；但是相信我，如果多花些时间来学习，阅读本书的其余部分时你便能够事半功倍。

我们先来谈谈内存地址，以及如何看待它们。指针代表内存地址，而内存地址归根结底只是个数字。所以，就像使用数字一样，你可以对指针执行一些数学运算。例如，指针与一个数相加，或两个指针相减。什么情况下你会这么做呢？比如当你想写一块内存，并且知道所要放置值的地方的实际偏移量时，就可以这么干。这一切听起来像天方夜谭吗？其实你已经多次这样用了，那就是——数组！

如下代码所示：

```
int x[ 10 ];
x[ 3 ] = 120;
```

你正在执行指针运算，将第3个内存槽位的值设置为120，方括号只是做指针运算的语法糖[①]（一个术语，意思是特殊的、简化的语法）。通过如下语句可以执行相同的操作：

```
*( x + 3 ) = 120;
```

我们来分析一下。令人惊讶而又迷惑的是，这不是把x的值增加3，而是增加了3 * sizeof(int)。sizeof是一个特殊的关键字，指以字节为单位返回一个类型的变量的大小，处理内存时经常会用到。指针运算总是加上内存“槽位”（slot），而不是直接加上数字（就像使用数组的方括号可以访问特定的数组槽位一样）。以变量大小为单位增减，能够防止意外使用指针对两个值之间的数据（例如，一个槽位的最后2字节和另一个槽位的前2字节）进行读写[②]。

大多数情况下，你应该使用数组语法，而不是试图进行正确的指针运算。做指针运算时，你很难一直清醒地知道正在进行的事情，很容易就忽略是在增加内存槽位而不是单字节。然而，理解指针运算能使你更容易做一些复杂的事情，后续几章将使用到这些能力。指针运算还有助于了解如何动态地分配多维数组。

14

---

① 语法糖实际上并没有给语言添加新东西，只是代码风格更好、更易读。——译者注

② 顺便说下，也可以让两个指针相减，以计算其距离。再次提醒，这个距离是槽位的数量而不是字节数（两个不同类型的指针不能相减，因为它们可能有不同大小的槽位）。我很少看到指针之间相减。永远不要把两个指针相加，因为你只可以将指针加上一个偏移量（指针相减或相加的结果是别的类型的值，这是不是很有趣呢）。

14.4.1　理解二维数组

在开始学习分配多维数组前，你需要知道多维数组的真正含义。再次提醒，这是一个哪怕困难重重也应该迎难而上努力理解的部分，守得云开见月明！

让我们从一个令人好奇的古怪现象谈起：当声明一个接收二维数组为参数的函数时，并不需要总是提供数组两个部分的大小，只要提供第二个的就行。

你可以两个大小都提供：

```
int sumTwoDArray( int array[ 4 ][ 4 ] );
```

或只提供第二个大小：

```
int sumTwoDArray( int array[][ 4 ] );
```

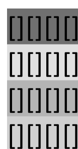
但你不能两个大小都省略：

```
int sumTwoDArray( int array[][] );
```

也不能只给出第一个大小：

```
int sumTwoDArray( int array[ 4 ][] );
```

这是为什么呢？因为只有某些大小已知，指针运算才能正确地进行。二维数组实际上是按先后顺序依次以条状存储在内存中的，编译器允许程序员把它当成一个正方形的内存块，但它其实只是地址的线性集合。编译器通过将数组访问（比如`array[ 3 ][ 2 ]`）转换成内存中的位置，来实现这一效果。有一个理解它的简单方法。如果你想象中的一个4×4的数组是这样子的：



但实际上，该数组在内存中是这样分布的：



为了使用`array[ 3 ][ 2 ]`（位于最后一组），编译器需要向下访问内存的三行（经过第一组、第二组和第三组这三行）和两列。由于要经过三行，而每行的宽度是4个整数，因此我们得走4×3个整数槽位，再加上两个整数槽位（为了到达最后一行中的第三个元素）。

换句话说，`array[ 3 ][ 2 ]`转换成了以下的指针运算：

```
*(array + 3 * <width of array> + 2)
```

现在可以看到，我们需要数组的宽度，没有它就不能完成计算。二维数组的第二维就是其宽度。数据放在内存中的物理方式，决定了仅有高度是不能计算出这个结果的（如果数组以另外一个方向来存放，这时候需要的可能就是高度了）。因此，当你把数组作为参数传递给函数时，数组的高度可有可无，但数组的第二维必须要明确指定。事实上任何的多维数组，都必须指定除高度之外的所有维度的大小。一维数组可以看做是数组的特例，即一个只有高度的数组。

问题又来了，由于声明二维数组需要宽度必须确定，动态分配一个具有任意宽度的二维数组就需要C++的一个特性——指向指针的指针。

14.4.2　指向指针的指针

除指向普通数据外，指针也可以指向其他指针。毕竟，指针就像其他任何变量一样，有一个可以访问的地址。

声明一个指向指针的指针，要这样写：

```
int **p_p_x;
```

p_p_x指向一个指针的内存地址，而这个指针又指向了一个整数。我使用前缀p_p标示指针本身指向另一个指针。这意味着，你需要给它提供一个指针的内存地址。比如：

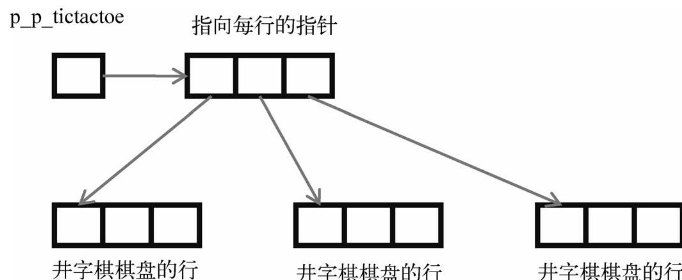
```
int *p_y;
int **p_p_x;
p_p_x = & p_y;
```

然后可以通过使用p_p_x将一个指针赋值给p_y：

```
*p_p_x = new int;
```

如同使用指针来创建一个任意大小的一维数组，我们可以以同样的方法，使用指向指针的指针来创建一个任意大小的二维数组。

可以这样思考，你有一个一维数组的指针，这些指针每一个都指向第二个一维数组。来看看示意图，假设我们声明了一个指向指针的指针来存储一个井字棋棋盘：



第一个指针，指向指针的集合，其中每个指针指向井字板的一行。以下是分配出这种数据结构的代码：

```
int **p_p_tictactoe;
// 注意，这里是int*，因为我们要分配一个指针数组
p_p_tictactoe = new int*[ 3 ];

// 现在，让每个指针都存储整数数组的地址
for ( int i = 0; i < 3; i++ )
{
    p_p_tictactoe[ i ] = new int[ 3 ];
}
```

这时可以像使用二维数组一样地使用分配的内存了。例如，我们可以用两个`for`循环来初始化整个井字板：

```
for ( int i = 0; i < 3; i++ )
{
    for ( int j = 0; j < 3; j++ )
    {
        p_p_tictactoe[ i ][ j ] = 0;
    }
}
```

要释放其内存，我们要按照同初始化完全相反的顺序来实行——首先释放每一行的指针，然后释放掉指向这些行的指针：

```
for ( int i = 0; i < 3; i++ )
{
    delete [] p_p_tictactoe[ i ];
}

delete [] p_p_tictactoe;
```

通常不会在已知内存大小时（比如创建井字板的情况）使用这种方法，下面这种写法要简单一些：

```
int tic_tac_toe_board[ 3 ][ 3 ];
```

但如果你想创建一个任意大的游戏板，应该用第一种写法。

14.4.3　指向指针的指针与二维数组

注意，使用指向指针的指针来存放二维数组时，这种二维数据在内存中的存放方式与普通的二维数组并不相同。一个标准的二维数组都是连续的内存，但基于指针的方法却不是。示意图中显示每一行都是一个单独的数据块。事实上，每一行在内存中的存储位置，可能会彼此完全远离。

将数组作为参数传递给函数，有一些需要注意的地方。你已经知道可以将一个数组赋值给指针：

```
int x[ 8 ];
int *y = x;
```

但是，不能将一个二维数组赋值给指向指针的指针：

```
错误代码
int x[8][8];
int **y = x; // 无法编译!
```

在第一种情况中，数组可以看做一个指针，它指向一块包含了所有数据的内存块。在第二种情况，数组仍然只是一个指向了一块内存块的指针。

这就是指向指针的指针与二维数组在内存中存放方式的不一样所导致的一个严重后果：不能将指向指针的指针传递给函数中的多维数组（尽管我们可以传递一个指针给函数中的一维数组）。

```
int sum_matrix (int values[][ 4 ], int num_rows)
{
    int running_total = 0;
    for ( int i = 0; i < num_vals; i++ )
    {
        for ( int j = 0; j < 4; j++ )
        {
            running_total += values[ i ][ j ];
        }
    }
    return running_total;
}
```

如下，将一个指向指针的指针传递给sum_matrix函数，编译器会报错：

```
错误代码
int **x;
// 分配一个指向指针的指针x为10行
sum_matrix( x, 10 ); // 无法编译
```

一维情况下，程序只是到指针地址的一个特定的偏移位置来取值进行操作。但在二维情况下，指向指针的指针这种方法需要用到两个指针引用：一个指针找到正确的行，另一个指针取出行中正确的值。而对于二维数组，只是使用指针的偏移运算来获得正确的值；由于一个指向指针的指针不能做这种运算，因此编译器不允许将指向指针的指针传递给二维数组，尽管你写的代码看起来是一样的！

14.5 盘点指针

学习指针一开始可能会让人觉得非常困惑，但你有能力理解它们。如果你还没有吃透指针的一切内容，那就多做几次深呼吸，然后重读本章，做完所有的测验，并努力解决实践题。不必做

到完全掌握所有何种情况下以及为何使用指针的每一个细微差别，但你应该知道初始化和使用指针的语法，并懂得如何分配内存。

14.6　问答题

(1) 下列哪一项是C++中分配内存的正确关键字？

A. `new`　　B. `malloc`　　C. `create`　　D. `value`

(2) 下列哪一项是C++中释放内存的正确关键字？[①]

A. `free`　　B. `delete`　　C. `clear`　　D. `remove`

(3) 以下说法哪一项是正确的？

A. 数组与指针是一样的
B. 数组不能够被赋值给指针
C. 指针可以被当做数组，但两者是不一样的
D. 可以像数组一样地使用指针，但不能将数组分配给指针

(4) 下面的代码中，`x`、`p_int`和`p_p_int`最终的值是多少？（注意，由于整数和指针是不同的类型，编译器不会直接接受此代码，但此练习是有用的，通过纸上的代码同样有助于理解多维指针。）

```
int x = 0;
int *p_int = & x;
int **p_p_int = & p_int;
*p_int = 12;
**p_p_int = 25;
p_int = 12;
*p_p_int = 3;
p_p_int = 27;
```

A. `x = 0, p_p_int = 27, p_int = 12`
B. `x = 25, p_p_int =  27, p_int = 12`
C. `x = 25, p_p_int = 27, p_int = 3`
D. `x = 3, p_p_int = 27, p_int = 12`

(5) 你怎样能表明一个指针没有指向有效的值？

A. 将指针置为负数　　B. 将指针置为`NULL`
C. 释放与指针相关联的内存　　D. 将指针置为`false`

---

① 好吧，如果这两题你回答的是`malloc`和`free`，也算对，这两个是从C延续过来的函数——但你可能没有阅读本章！

14.7 实践题

(1) 写一个函数，创建一个二维乘法表（http://math2.org/math/general/multiplytable.htm），它的两个维度的大小不固定。

(2) 写一个函数，它接受三个参数，分别是`length`、`width`和`height`，用这三个值动态地分配一个三维数组，并用乘法表填充此三维数组。确保必要时释放内存。

(3) 写一个程序，输出二维数组中每一个元素的内存地址。验证输出值是否与本书解释的存储方式一致。

(4) 写一个程序，让用户跟踪他们最近一次跟每个朋友交谈的时间。用户应该能够添加新朋友（数量没有限制）并保存最近一次跟朋友交谈距今的天数。天数由用户更新（但是不能输入无效的数值，比如负数）。确保列表可以按照名称和时间排序进行显示。

(5) 写一个双人玩的四子棋游戏①。用户可以设置棋盘的长和宽，每个玩家轮流落子。在棋盘中分别用+和x表示双方的棋子，用_表示空位。

(6) 写一个程序，输入宽（`width`）和高（`height`），根据这两个值动态地产生一个迷宫。迷宫必须始终有一条能够通过它的有效路径（想想要怎么保证这一点）。迷宫生成后，输出到屏幕中。

所有的实践题，尝试分别写一个指针的版本和一个引用的版本。确保释放掉所有你分配的内存。

① 参见http://en.wikipedia.org/wiki/Connect_Four。

第 15 章 数据结构简介与链表

上一章介绍了如何通过分配内存动态地创建数组，这一章讲解如何更灵活地使用动态内存分配。拥有大量内存最大的好处是，你会有很多位置来存放数据，可以存储相当多的东西。但接下来的问题是：要怎么做才能迅速进行存储，并便捷地访问到这些数据呢？本章就是要讨论这个问题。

首先介绍术语*数据结构*，它是指内存中组织数据的一种方式。例如，数组就是一个非常简单的数据结构，它以线性方式组织内存中的数据。数组中的各个元素就是数据结构中的各个元素。使用指向指针的指针实现的二维数组是一个较为复杂的数据结构。

问题是，使用数组时，如果该数组没有空位，就不能添加数据到该数组中。此时，你就必须从头开始，重新分配一个数组，然后将现有数组中的所有元素都复制到新数组中。这就是计算机程序员所说的*昂贵操作*——以计算机处理器的标准，这会花费很长时间。但从用户的角度看，这实在没什么大不了的：计算机处理器的速度已经相当快，如果这种操作不是经常进行，没有人会注意到有什么问题。但有时候，“昂贵操作”会带来严重的麻烦。

关于数组的第二个问题是，你不能轻易地在现有的数组元素之间插入数据。举个例子，如果想在第一个元素和第二个元素之间插入一个新元素，而数组有1000个元素，那么必须把元素2至元素1000各向后挪一位！这个操作也很昂贵。

有时候会遇到这种情况——电脑嘎嚓嘎嚓地响，让你痛苦地等待，怎么回事呢？这就是因为的电脑在做某些昂贵操作。数据结构正是为了减轻这些问题，而创造出来的一种进行高效的数据存储的方式，让用户不用再盯着死亡的沙滩球[①]发呆。

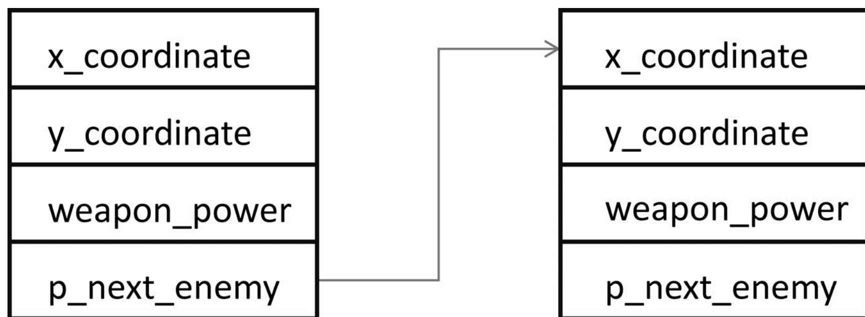
使用不同的数据结构的第二个原因是，它可以使你站在更高的层次来思考编程问题。与其总是讨论烦琐的“循环”，我们开始从更高的层面开始讨论“列表”。数据结构提供了组织数据的一种逻辑方式，以及交流程序所用到的基本操作的一种简便说辞。比如说你需要一个“列表”，就

① 死亡的沙滩球，也称为“彩虹轮”，是苹果Mac OS X中的一种图标，用于指示应用程序没有响应系统。
——译者注

很清楚地表达了：需要将数据以一种能够高效地增加和删除的方式进行存储。了解了更多数据结构后，你会逐渐学会从数据角度思考程序，学会思考如何组织数据。好了，不谈那些理论的东西了，让我们先来谈谈链表吧。

还记得那个问题吗：怎样简便地增加一个数据元素？用一个必须复制一遍原数组的数组来实现，感觉如何？（但愿你还记得，我们刚刚讨论过这个问题。）试想一下，如果有这样一个数据结构：其中的每一项数据都能告诉你去哪里找下一项数据，会不会很赞？这样的话，我们就可以通过让最后一个元素指向新添加的元素，轻而易举地在原有的数据结构末尾添加了新元素。如果要在两个元素间插入新元素，也很简单：仅仅通过改变这两个元素之一的指向即可。回忆下以前使用过的例子，存储游戏中的敌舰数据。很容易想到将敌舰以某种形式的列表来存储，该列表中的每一个元素是一个存储了敌舰信息的结构体。为什么要将敌舰数据构成一个列表呢？你会想要的，如果需要在每一轮游戏中都对所有的敌舰采取行动。比如你想遍历列表中的所有敌舰，让每个敌舰移动下位置，就需要这个列表。这不是一个像购物清单或班级学生花名册一样的“列表”，有时你只是用列表形式存储下自己拥有的所有东西，就好像上述例子里的敌舰一样。另外，你还可能会希望能够快速地添加、删除敌舰。那么，如果让每个敌舰都具有下一部敌舰的信息，会怎样？

先看一下下面的示意图。可以直观地看到，每个敌舰元素都包含其在屏幕的*x*坐标、*y*坐标，以及一定的武器火力值：



我们用结构体`EnemySpaceShip`来存储每个敌舰元素，其中，每个结构体都具有一个到下一个结构体的链接。这个链接会是什么呢？对的，指针！每艘敌舰都有一个指向下一艘敌舰的指针：

```
struct EnemySpaceShip
{
    int x_coordinate;
    int y_coordinate;
    int weapon_power;
    EnemySpaceShip* p_next_enemy;
};
```

等一下！结构体`EnemySpaceShip`的定义中又使用了结构体`EnemySpaceShip`，这真的可以吗？是的，没问题！C++完全有能力处理这种自我引用。你如果在结构体里写成下面这样才会出问题：

```
EnemySpaceShip next_enemy;
```

这样就形成了一个无限重复自身的结构体。声明这样的一艘敌舰将会耗光系统的所有内存。因此请注意，我们写的是一个指向 EnemySpaceShip 的指针，而不是一个真实的 EnemySpaceShip。由于指针不一定要指向有效内存，所以这样写不会产生一个敌舰的无限列表，而只是产生了一艘敌舰，这艘敌舰“有可能”会指向另一艘敌舰。如果它指向的另一艘敌舰是真实的，那么这另一艘敌舰当然会占用一些额外的内存，但在这之前，该结构体只多占用了一点点内存来存放指针——只有几字节而已。指针仅仅是表示它有可能指向一块有效内存，指针本身的存储只需要一小块内存空间而已。当声明一个EnemySpaceShip时，你需要足够的空间来容纳 x_coordinate、y_coordinate、weapon_power，以及最后的指针。你不需要容纳另一艘敌舰，只要容纳一个指针就可以了。

打个比方吧。想象一列火车，火车上的每一节车厢都有一个钩子，可以用来勾住另一节车厢。要添加一节新车厢，你只需要把新车厢和它前面及后面的车厢都连接上即可。如果没有车厢要连接，钩子也可以不用，这个钩子就相当于一个空指针。

我们已经讲述了生成这些类型列表的相关概念，现在来学习使用指针与结构体的一些细节和语法。

15.1 指针和结构体

要通过指针访问结构体的域，可以在使用“.”运算符的位置使用“->”运算符：

```
p_my_struct->my_field;
```

结构体的每个成员变量具有不同的内存地址，通常距离结构体的起始地址若干字节——箭头语法能用来计算出这个偏移量。对箭头语法而言，指针的其他所有属性仍然适用（比如一个指针指向的是一块内存、不能使用无效指针，等等），它完全等价于下列写法：

```
(*p_my_struct).my_field;
```

但是，箭头语法更便于阅读，熟练掌握的话，书写也很方便。

如果一个函数接受指向结构体的指针为参数，那么，它就能够修改与该结构体相关联的内存地址。也就是说，我们允许函数修改传入进来的结构体，这实际上跟把数组传递到函数中的情况是类似的。来看看在结构体EnemySpaceShip中是怎样处理的：

```
// 头文件<cstddef>用于NULL，通常包含在其他头文件中，
// 但这里不需要使用任何其他的头文件，所以直接include这个<cstddef>了
#include <cstddef>

struct EnemySpaceShip
{
```

```
    int x_coordinate;
    int y_coordinate;
    int weapon_power;
    EnemySpaceShip* p_next_enemy;
};

EnemySpaceShip* getNewEnemy ()
{
    EnemySpaceShip* p_ship = new EnemySpaceShip;
    p_ship->x_coordinate = 0;
    p_ship->y_coordinate = 0;
    p_ship->weapon_power = 20;
    p_ship->p_next_enemy = NULL;
    return p_ship;
}

void upgradeWeapons (EnemySpaceShip* p_ship)
{
    p_ship->weapon_power += 10;
}

int main ()
{
    EnemySpaceShip* p_enemy = getNewEnemy();
    upgradeWeapons( p_enemy );
}
```

示例代码38：upgrade.cpp

在getNewEnemy中，我们使用new来为一艘新敌舰分配新内存。在upgradeWeapons中，由于p_ship指向了一块包含结构体的所有成员变量的内存空间，因此我们能够修改结构体p_ship。

15.2　创建一个链表

现在，我们已经掌握了结合使用指针和结构体的语法，可以创建自己的列表了。任何时候，我们通过使用包含一个指针指向下一个元素的结构体所创建出来的列表，都称为链表（linked list）。为了总能找到这个链表，我们需要以某种方式，来记住该链表的起始位置。回到刚才的例子，可以给敌舰链表增加一个指向其起始点的指针：

```
struct EnemySpaceShip
{
    int x_coordinate;
    int y_coordinate;
    int weapon_power;
    EnemySpaceShip* p_next_enemy;
};

EnemySpaceShip* p_enemies = NULL;
```

p_enemies是一个指针变量，指向整个敌舰列表。每当我们在游戏中增加一艘敌舰，就将该敌舰添加到此列表中。（这个p_enemies将是在游戏中对所有敌舰做某件事时的出发站点。）这个变量也可以命名为p_first或p_head，以表明它是列表的第一个元素。

每当在游戏中添加一艘新敌舰，我们都将它添加到列表的最前面：

```
EnemySpaceShip* getNewEnemy ()
{
    EnemySpaceShip* p_ship = new EnemySpaceShip;
    p_ship->x_coordinate = 0;
    p_ship->y_coordinate = 0;
    p_ship->weapon_power = 20;
    p_ship->p_next_enemy = p_enemies;
    p_enemies = p_ship;
    return p_ship;
}
```

开始时，p_ship为空（NULL）。每当创建一艘新敌舰，我们都要更新这艘新敌舰，使其指向链表的第一个元素（存储在p_enemies中），接着使p_enemies指向新建立的敌舰。这相当于使列表中的其余元素向后“滑动”一位，让新元素添加到了列表的前面。这里的“滑动”并不需要任何的复制操作，我们只是修改了两个指针。

这可能有点令人不解。所以我们通过一系列步骤以及示意图来加深理解。

15.2.1 第一轮

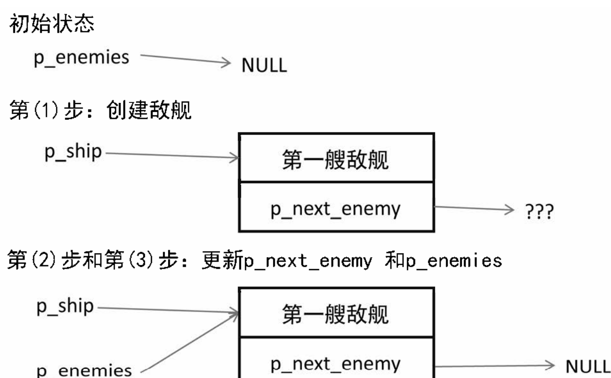
在初始状态下，p_enemies一开始便为NULL。换句话说，此时没有敌舰（我们总是用NULL来表示列表的末尾）。

(1) 分配一艘新敌舰，用指针p_ship指向它。现在有了一个新的敌舰，我称为SHIP1，它现在还未存储在列表中。在图中可以看到，p_next_enemy尚未确定，它指向未知的内存。

(2) SHIP1的成员变量p_next_enemy设置为指向当前的敌舰列表（在本例中为NULL）。

(3) 更新p_enemies为指向新创建的敌舰。

(4) 函数返回p_ship给调用者，进行任何所需的使用，而p_enemies提供了访问整个列表的入口（当前，列表只有一个元素）。



15.2.2 第二轮

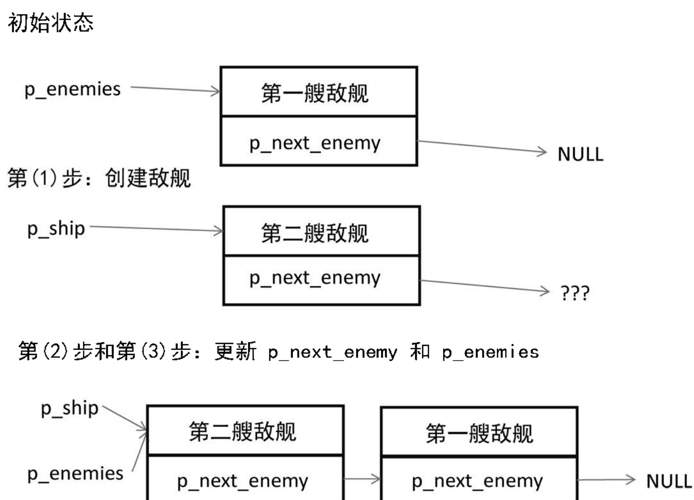
第二轮刚开始时，p_enemies指向着刚创建的敌舰。

(1) 分配一艘新的敌舰，用指针p_ship指向。现在有了第二艘敌舰，它的p_next_enemy指向未知的内存。

(2) p_next_enemy被设置为指向当前的敌舰列表。在本例中，p_next_enemy指向我们第一轮中创建的敌舰。

(3) 更新p_enemies指向最新创建的敌舰（p_enemies现在指向了第二艘敌舰，而第二艘敌舰指向第一艘敌舰）。

(4) 函数返回p_ship给调用者，进行任何所需的使用，而p_enemies提供了访问整个列表的入口（当前，列表中有两个元素）。



每当插入一个新元素时，你可以将这些操作视为向下“滑动”列表中所有的已有元素。这种“滑动”不像数组中的那样，需要复制整个列表，而只需要更新指向列表起始点的指针，使其指向新的开始元素。列表中的第一个元素称为列表的头节点。通常要有一个指针指向列表的头结点，在本例中是p_enemies。函数结束时，p_ship和p_enemies指向了同一位置，在这之前，我们需要用指针p_ship抓住新分配的内存，这样就可以修改新节点的p_next_enemy指向存储在p_enemies中的列表的头节点。

虽然在刚才的函数中，我们将链表的头指针作为全局变量来使用，但你也可以把它作为参数传给函数，这样，这个函数就能处理任何链表，而不只是一个全局链表了。以下是可行的代码：

```
EnemySpaceShip* addNewEnemyToList (EnemySpaceShip* p_list)
{
    EnemySpaceShip* p_ship = new EnemySpaceShip;
    p_ship->x_coordinate = 0;
    p_ship->y_coordinate = 0;
    p_ship->weapon_power = 20;
    p_ship->p_next_enemy = p_list;
    return p_ship;
}
```

注意，addNewEnemyToList返回的是指向链表的指针，而不是新创建的敌舰的指针，这与getNewEnemy的做法不同。由于addNewEnemyToList中既没有与列表相关联的全局变量，也无法修改传递到函数中的链表头指针（只能修改指针所指向的东西），因此还需要一种方式来告知调用者列表的新起始点[①]。调用者的代码可以这样写：

```
p_list = EnemySpaceShip* addNewEnemyToList( p_list );
```

函数addNewEnemyToList的接口让其调用者可以选择所使用的列表以及在何处存储返回的列表。

用addNewEnemyToList函数也可以模拟之前p_enemies为全局变量的函数的行为，你可以这样写：

```
p_enemies = EnemySpaceShip* addNewEnemyToList( p_enemies );
```

15.3　遍历链表

到目前为止，一切都很好。现在知道了如何在列表中存储东西，使用列表来做些实实在在的事情吧。但做点什么好呢？我们都知道如何使用for循环来迭代地访问数组的每个元素（迭代其实是循环的一种高端说法）。来学习如何对链表做同样的事情，即遍历链表。

---

① 想要给自己一个真正的头脑训练吗，试试使用指向指针的指针而不是返回原来的值来解决同样的问题吧。

要取得列表中的下一个元素，只需要知道当前元素即可。你可以写一个循环，其中有一个变量持有指向列表的当前元素的指针，每当对当前元素执行操作后，就更新它指向列表的下一个元素。

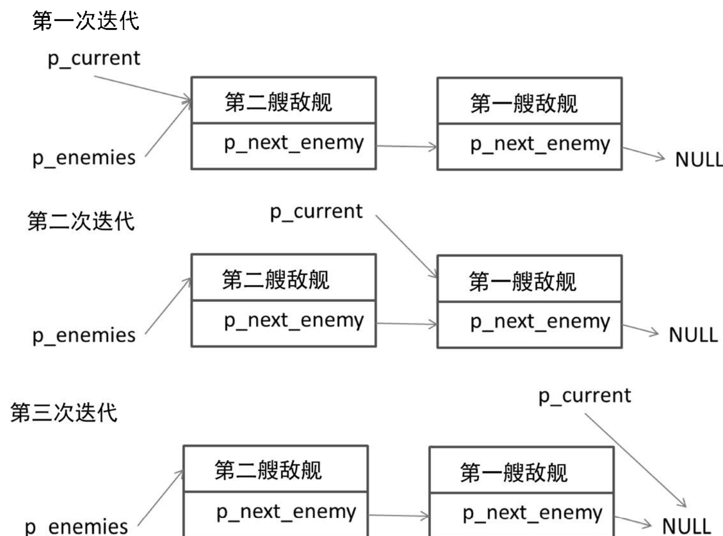
来看一段示例代码，它将升级游戏中所有敌舰的武器（可能因为玩家升到了下一级）：

```
EnemySpaceShip *p_current = p_enemies;

while ( p_current != NULL )
{
    upgradeWeapons( p_current );
    p_current = p_current->p_next_enemy;
}
```

哇，代码行数跟遍历数组的几乎一样短呢！变量p_current用来跟踪列表的当前元素，初始时，它指向列表的第一艘敌舰（无论p_enemies指向何处）。只要p_current不为NULL（意味着还没到列表的末尾），我们就升级当前敌舰的武器，并更新p_current指向列表中的下一艘敌舰。

请注意，整个程序中，只需要简单改变p_current的指向，而p_enemies和其他指针仍指向相同的位置。这就是指针的强大之处！它让你仅通过修改指针的指向就可以沿着整个数据结构移动，不需要任何复制操作。任何时候，每艘敌舰都只有一个副本。这使得我们的武器升级代码能够修改列表中原来的敌舰，而不是修改敌舰的副本。下图是迭代遍历列表的过程中，数据结构和变量的可视化表示：



15.4　盘点链表

链表允许轻松地添加新的内存到数据结构中，而没有大量的内存复制和数组拼凑。你还可以实现一些其他操作，比如添加元素到列表中间或删除元素。一个链表的完整实现，应该提供上述所有操作。

告诉你一个小秘密，你可能永远不会需要实现自己的链表！你可以使用标准模板库，而不是自己写一个链表。我们很快就会讨论到标准模板库。然而，链表的重要之处在于，我们经常会使用类似的技术来创建更有趣的数据结构。相信我，我没有让你误入歧途——这里学到的东西一定会有价值，即使你从来不编写自己的链表。此外，通过了解一个链表是如何实现的，可以更好地理解使用链表和数组的利弊。

数组和链表的比较

链表优于数组的主要地方在于，链表可以轻松地调整大小或添加元素，而且这样做不需要移动每个元素。例如，你很容易将新节点插入到链表中。

如果你想将新元素插入到一个排好序的列表中，同时要保持列表元素的顺序不变，会怎样呢？假设这个列表为1、2、5、9、10，你想将元素6添加到5和9之间。对于数组来说，需要调整数组大小，以便能够容纳新元素，因此你必须移动从9到列表末尾的每个元素。如果列表在10之后有1000个元素，你必须将它们统统向后移动一位。换句话说，将元素插入到数组中的性能，与数组的长度是成比例的。如果用的是链表，你只需修改元素5指向新的元素，修改新元素指向元素9，就大功告成了！无论列表有多大，插入操作需要的时间都一样。

数组优于链表的地方主要在于，在数组中选择任意一个元素都很快，只需提供该元素的索引即可。而对于链表来说，需要遍历链表中的每个元素，直到查找到想要的元素为止。不过，数组的这一优势，建立在数组的索引与元素中存储的值之间有关联的基础上。否则，你还是得通过遍历数组来找到想要的元素。

例如，你可以用数组来创建一个选票计数器。其中，候选人从0到9进行编号，选民使用数字0~9来投票。然后，每个数组索引对应一个候选人，数组在该位置上的值即是候选人的票数。候选人与这些数字之间并没有内在联系，但我们通过对候选人编号简单地建立了一种联系。接着我们使用这些数字来获得候选人的信息。

下面是使用数组的一个实现：

```
#include <iostream>

using namespace std;

int main ()
```

```
{
    int votes[ 10 ];

    // 确保选举没有作弊（通过清空数组）
    for ( int i = 0; i < 10; ++i )
    {
        votes[ i ] = 0;
    }

    int candidate;
    cout << "Vote for the candidate of your choice, using numbers: 0) Joe
1) Bob 2) Mary 3) Suzy 4) Margaret 5) Eleanor 6) Alex 7) Thomas 8) Andrew 9)
Ilene" << '\n';
    cin >> candidate;

    // 输入选票，直到用户输入一个非候选人编号
    while ( 0 <= candidate && candidate <= 9 )
    {
        // 注意，不能使用do-while循环，
        // 因为需要在更新数组之前检查下candidate是否在正确的范围内
        // 一个do-while循环将需要读入candidate的值，
        // 然后再进行检查，接着增加对应的票数
        votes[ candidate ]++;
        cout << "Plese enter another vote: ";
        cin >> candidate;
    }
    // 显示票数
    for ( int i = 0; i < 10; ++i )
    {
        cout << votes[ i ] << '\n';
    }
}
```

示例代码39：vote.cpp

看，更新某个特定的候选人票数多么容易呀！

我们可以做得更漂亮：保持一个结构体数组，每个结构体包含票数和候选人姓名。这种方法可以很容易地输出票数和候选人姓名。

试想一下，如果你试图用链表做同样的事情会怎么样呢？该代码必须一个元素接着一个元素地前进，直到抵达所选择的候选人为止。增加一张候选人5的票数，需要不断循环，从候选人0的节点走到候选人1的节点，接着到候选人2的节点，而没办法跳到链表的中间。

通过索引访问数组的一个元素所花费的时间是恒定的，这意味着它不因数组的大小而变化。相反，找到链表中的一个元素所花费的时间，则是与列表的大小成正比。随着列表大小的增长，这将变得越来越慢。

如果打算使用链表来做的话，给候选人编号就没什么意义了，你也可以改为通过候选人姓名查找。（姓名比较与索引比较相比速度要慢，但既然选择了使用链表来实现，也许你不是很在乎

代码的效率。）

1. 链表需要的存储空间大小

元素数量很大时，数据结构所占用的空间大小是一个不得不考虑的衡量因素。对于小型的数据结构，其差别并不大；但是，如果数据结构本身就比较庞大，两倍的占用空间量可能是一个大问题。

数组的每个元素通常占用的空间较少。链表需要列表中的元素和一个指向列表的下一个元素的指针。这意味着链表从一开始就需要大约两倍于每个元素的空间。不过，如果事先不知道要存储的元素数目的话，有时链表占用的空间反而比数组少。与其分配一个大数组，然后让许多的数组元素空着，不如只在需要时才分配新的链表节点，这样就不会浪费没有使用到的额外空间。（为了避免这个问题，可以动态地分配数组，但是这需要在每次分配更多内存时复制数组元素，从而抵消了一些占用空间小的优势。[①]）

2. 其他考虑因素

数组也可以是多维的，例如，用数组很容易就能表示一个8乘8的棋盘。然而，要用链表来表示这个棋盘就需要一个包含其他列表的列表，这使访问特定元素的速度慢了许多，而且更难理解。

3. 一般的经验法则

下面是两个关于何种情况下应该使用链表，何种情况下应该使用数组的经验法则：

(1) 当需要通过索引以常量时间访问元素，且预先知道需要存储多少元素，或当需要尽量减少每个元素所占用的空间时，建议使用数组；

(2) 当需要能够不断地增加新元素[②]，或需要在列表的中间做大量的插入时，建议使用链表。

换句话说，链表和数组各有优点，选择使用链表还是数组取决于你想做什么。

15.5　问答题

(1) 链表相比于数组的优势是？

A. 链表的每个元素占用空间较少

B. 链表可以动态地扩展新元素而不用复制已有元素

C. 链表可以更快地找到特定元素

---

① 不管怎样，你可以选择这种做法，特别是如果想要通过索引，在常量时间里访问到数组元素时。对于数据结构，当你在比较几个不是明显糟糕的解决方案时，通常没有普遍正确的答案。

② 标准模板库（STL）中的vector类实际上使得增加元素到类数组的数据结构中变得很容易，从而使链表的这一优势不再明显。因此，vector是一个比链表和数组更好的选择。稍后我们将讨论vector。

D. 链表可以将结构体容纳为元素

(2) 以下哪项是正确的？

A. 没有任何理由使用数组
B. 链表和数组具有相同的性能特点
C. 链表和数组都允许按索引以常量时间访问元素
D. 在链表中间插入元素比在数组中间插入速度要快

(3) 通常什么时候使用链表？

A. 只需要存储一个元素时
B. 需要存储的元素个数在编译时刻已知时
C. 需要动态地添加和删除元素时
D. 需要快速访问已排序的列表中的任何一个元素而无需做任何迭代时

(4) 为什么声明一个引用了自身元素类型（`struct Node { Node* p_next; };`）的链表不会有问题？

A. 这是不允许的
B. 因为编译器能够弄清楚你实际上并不需要自引用元素的内存
C. 因为该类型是一个指针，你只需要足够的空间来容纳一个指针，实际的下一个节点的内存之后才会分配
D. 只有你实际上不分配`p_next`指向另一个结构体时才允许这么做

(5) 为什么在链表末尾有一个空指针（`NULL`）很重要？

A. 它指示了链表结束的位置，防止代码访问未初始化的内存
B. 它能防止列表循环引用
C. 它能帮助调试，如果你试着偏离列表太远，程序将崩溃
D. 如果我们不存储`NULL`，那么列表将因为自引用而需要无限的内存

(6) 链表和数组的有什么相似性？

A. 两者都允许你快速地在当前列表的中间添加元素
B. 两者都允许顺次地存储数据，并顺次访问数据
C. 两者都可以通过添加元素而变得更大
D. 两者都提供了对列表中的每个元素的快速访问

15.6 实践题

(1) 写一个程序，它能将元素从一个链表中删除；删除函数应该只移除要删除的元素。这个

函数好写吗？能否通过在列表中添加额外的指针，使得程序更容易写或速度更快？[①]

(2) 写一个程序，它能将元素添加到有序的链表中，而不是添加在链表的开头。

(3) 写一个程序，它能通过名称找到链表中的元素。

(4) 实现一个双人的井字棋游戏。先是使用链表来表示棋盘，再试着使用数组。哪个比较容易些？为什么？

① 提示：如果有一个指向前一个节点的指针，是否会好点？

第 16 章

递　　归

16

你已经见过许多基于循环的算法，它们一遍又一遍地执行某些任务。现在来讲另一类不使用循环却可以重复执行代码的方法，这种方法使用的是重复的函数调用，我们把它称为*递归*。递归是一种在表达操作时会用到自身的技术，也就是说，递归意味着编写的函数会调用自身。它跟循环类似，但功能更强大。它可以使某些几乎不可能用循环来完成的程序变成小事一桩！递归尤其适合于应用在诸如链表、二叉树（马上就讲到了）这样的数据结构中。接下来的两章内容，我们一起通过一些具体的例子，来探讨递归的基本思想。

16.1　如何看待递归

一个思考递归的有效方法是：把递归看做一个执行过程，这个执行过程的其中一条指令是“重复这个执行过程”。这听起来跟循环非常类似，因为都是在重复相同的代码。递归和循环确实在某些方面是类似的，但是，递归可以更容易地表达这样一种想法：执行过程的结果是完成执行过程所必需的。当然，这个“执行过程”必须存在某个时刻可以不用再递归调用就能够完成。举个简单的例子，砌筑一面十尺高墙。如果我想建造一面10英尺高的墙，我会先建造一个九英尺高的墙，然后添加一层额外的墙砖。从概念上讲，这就好比说：“建墙”函数接受了一个高度值，如果这个高度值大于1，“建墙”函数首先要调用自身来建造一个稍低的墙，然后添加一层额外的墙砖。

这个“建墙”函数的基本结构看起来应该如下面的代码所示。（这段代码有几个明显的缺陷，我们很快会讨论到。）这里面最重要的思想是：建造一个特定高度的墙可以用建造一个更低的墙来表达。

```
void buildWall (int height)
{
    buildWall( height - 1 );
    addBrickLayer();
}
```

但这段代码有一个小问题，不是吗？什么时候会停止调用buildWall呢？很遗憾，答案是，

永远不。解决办法很简单：我们需要在墙高为0时停止递归调用。墙的高度为0时，我们应该仅仅添加一层墙砖即可，不用建造任何更低的墙体。

```
void buildWall (int height)
{
    if ( height > 0 )
    {
        buildWall( height - 1 );
    }
    addBrickLayer();
}
```

函数不调用自身的情况称为函数的基线条件[①]。在刚才的例子中，“建墙”函数知道如果已经到达地面，就只要添加一层墙砖就可以了（建墙的基线条件）。否则，我们仍然需要建立一堵更低的墙，然后在上面添加一层砖。如果你对这段代码还是疑惑不解（第一次见到递归时，人们往往一头雾水），想想建造一堵墙的物理过程。刚开始，你希望建造一堵特定高度的墙，接着就会说：“我需要一堵矮一层的墙，好让我把砖块放上去。”最终，你就会说：“我不需要一堵更矮的墙了，我可以直接在地面上建造。”这就是基线条件。

注意，这个算法先将一个大问题简化成更小的问题（建造一堵更矮的墙），然后去解决这个更小的问题。在某些情况下，更小的问题（如在地面上建造一层高的墙体）小到不再需要进一步简化，而是可以马上就解决。在现实生活中，这意味着可以建立一堵墙了；而在C++里，这确保了该函数将最终停止递归调用。这很像之前看到过的自顶向下的设计过程，我们把问题分解成更小的子问题，创建出这些子问题的函数，然后用它们来构建完整的程序。这种情况下，我们将问题分解成了不同的子问题，而不是一个正在解决的问题；而在递归中，我们将一个问题分解成了相同问题的更小版本。

一旦函数调用了自己，当调用返回时，它会去执行调用点之后的下一行语句。类似的，递归调用返回后，函数仍可以执行操作或调用其他函数。在“建墙”的例子中，建造小墙后，函数将继续执行，添加一层新的砖块。

下面是一个实际可运行的例子，用来展示实际的输出。怎样写出一个递归函数，来输出数字123 456 789 987 654 321呢？我们可以先编写一个函数，它接受一个数字，然后两次输出这个数字，一次在函数递归之前，一次在递归之后。

```
#include <iostream>

using namespace std;

void printNum (int num)
{
    // 函数的两次cout调用，将像“三明治”一样输出
```

① 有时也称为终止条件。——译者注

```
    // 形如 (num+1)...99...(num+1) 的数字序列
    cout << num;
    // 只要num小于9，就递归输出
    // 序列 (num+1) ... 99 ... (num+1)
    if ( num < 9 )
    {
        printNum( num + 1 );
    }
    cout << num;
}

int main ()
{
    printNum( 1 );
}
```

示例代码40：printnum.cpp

printNum函数的递归调用输出序列(num+1)...99...(num+1)。通过在调用printNum(num + 1)的前后两边各输出一次num，我们有效地创建了一个“三明治”：num被输出在(num+1)...99...(num+1)的两边，因而构成了序列(num)(num+1)...99...(num+1)(num)。如果num为1，最终将得到123 456 789 987 654 321。

你也可以这样来理解整个过程：printNum函数每次输出数字后会再次调用printNum函数，结果是先依次输出了1~9。当基线条件满足时，printNum将返回到每个递归调用，以函数返回的顺序再次输出数字。最后一个函数调用的值为9，由于达到了基线条件，它将立即输出数字9，而不是再次调用函数。

当num为9的函数返回时，它将返回到num为8的函数进行递归调用的位置，接着num为8的函数接着输出数字8，然后返回；接着num为7的函数继续执行，以此类推，直到完成所有的递归调用，返回到第一个递归调用的位置，此时num为1；接着输出数字1，任务完成。

16.2　递归和数据结构

有些数据结构会借用到递归算法，因为这些数据结构的组成可以描述成含有相同数据结构的更小版本。既然递归算法通过将问题分解成原问题的更小版本来解决，数据结构也一样可以将原数据结构分解成相同数据结构的更小版本——链表就是一种这样的数据结构。

之前已经说过，链表是这样一种列表：你可以在链表前面增添更多的新节点。但从另一个角度去思考，也可以认为，链表由一个首节点构成，这个首节点指向了另一个更小版本的链表。

这一点很重要，因为它提供了一个非常有用的特性：可以编写这样一种处理链表的程序，它要么处理当前节点，要么去处理“列表的其余部分”。例如，要找到列表中的一个特定节点，可以使用此基本算法：

如果我们在列表的末尾，返回NULL。
否则，如果当前节点就是查找的目标，将其返回。
否则，在列表的其余部分继续查找。

在代码中，应该是这样的：

```
struct node
{
    int value;
    node *next;
};

node* search (node* list, int value_to_find)
{
    if ( list == NULL )
    {
        return NULL;
    }
    if ( list->value == value_to_find )
    {
        return list;
    }
    else
    {
        return search( list->next, value_to_find );
    }
}
```

当考虑一个递归调用时，我们提到过，被调用函数中会做一些事情。函数在给定的输入下所承诺要做的事，称为函数的契约。函数契约总结了函数所要做的事情。search函数的契约是查找到列表中的一个给定的节点。search函数的实现就相当于在说，“如果当前节点是我们想要找的，那么返回它；否则，函数的契约还是在列表中查找某个节点，让我们用这个契约，来看看剩余的列表吧！”

在列表的剩余部分调用search函数，而不是整个列表，这一点很重要。

递归只有在满足以下两个条件时，才能够正确运行：

(1) 能够构造出一个通过解决同类型的较小问题来解决原问题的方案；
(2) 能够解决基线条件。

search函数的解决有两个可能的基线条件：要么到达列表的末尾，要么找到想要的节点。如果这两种情况都没有满足，那么使用search函数来解决相同问题的较小版本。关键在于：我们能够递归地利用相同问题的较小版本的解决结果，来解决更大的原问题，只有这样，递归才能起到效果。

有时候，递归调用的返回值并不是马上被返回，而是被实际使用。让我们来看一个例子：数

学上的阶乘函数。（人们超爱用阶乘来作递归的例子！）

```
Factorial( x ) = x * ( x - 1 ) *( x - 2 )...*1
```

或者，换种方式来表达：

```
Factorial( x ) =
    If ( x == 1 ) 1
    Else x * Factorial( x - 1 )
```

换句话说，任何数字的阶乘就是此数字乘以比此数字小1的数的阶乘。在这种情况下，我们可以用递归调用的返回值做其他事情，比如将返回值乘以当前的数字，如下所示。

代码可以这样写：

```
int factorial (int x)
{
    if ( x == 1 )
    {
        return 1;
    }
    return x * factorial( x - 1 );
}
```

这个例子中，我们要么抵达x为1的基线条件，要么继续解决更小版本的相同问题，也即factorial(x - 1)，然后使用factorial(x - 1)的返回值来计算x的阶乘。每一次调用factorial都将使x变得更小，所以最终肯定会到达基线条件。

请注意，使用递归的过程中，我们不断地求解子问题，然后用子问题的结果来做一些事。在搜索一个链表时，我们只是返回子问题的求解结果。递归用于两种方式：要么是仅靠递归调用就能够解决全部的问题，要么是获得子问题的求解结果，然后使用该结果做更多的计算。

16.3　循环和递归

在某些情况下，递归算法可以很容易地转化成用结构相同的循环来表示。例如，搜索列表的代码可以写成这样：

```
node *search (node *list, int value_to_find)
{
    while ( 1 )
    {
        if ( list == NULL )
        {
            return NULL;
        }
        if ( list->value == value_to_find )
        {
```

```
            return list;
        }
        else
        {
            list = list->next;
        }
    }
}
```

这段代码进行的检查实际上跟使用递归的版本是一样的，你很容易看出两者的差异。两种算法的唯一区别是，这段代码使用了一个循环，而不是递归。它没有使用递归调用来缩短列表的大小，而是通过每次将它指向“列表的剩余部分”来实现的。这是一个递归的解决方案和迭代（基于循环）的解决方案有相似之处的例子。

当不需要对递归调用函数的返回值做任何处理时，通常很容易写出递归算法的循环版本，反之亦然，我们也能很容易写出循环算法的递归版本。这种情况就是*尾递归*（tail recursion）：递归调用是递归函数在函数尾部所做的最后一件事情。由于递归调用是最后一个操作，这无异于循环中的下一步。一旦下一个调用完成，之前的调用就不再需要了。列表搜索就是一个尾递归的例子。

然而，如果考虑将递归实现的阶乘函数改写成基于循环的实现时，问题就出现了：

```
int factorial (int x)
{
    while ( 1 )
    {
        if ( x == 1 )
        {
            return 1;
        }
        // 要返回 x * factorial( x - 1 );
        // 以下代码应该怎么写呢?
    }
}
```

我们需要用`factorial( x - 1 )`来做一些事，所以这里不能只是循环。也就是说，我们需要真正地解决了子问题，才能够完成计算。

其实，如果重新从另外一个角度来思考，就会发现：阶乘函数很容易转换成用循环来实现。来考虑一下原来的定义：

```
Factorial( x ) = x * ( x - 1 ) *( x - 2 )...*1
```

如果能跟踪当前值，就可以通过将运行中`x * ( x - 1 ) * ( x - 2 ) ...`乘法运算的结果存储下来，从而解决问题：

```
int factorial (int x)
{
```

```
    int cur = x;
    while ( x > 1 )
    {
        x--;
        cur *= x;
    }
    return x;
}
```

注意，我们并非通过获得子问题（更小的阶乘）的结果来解决这个问题，而是以相反的顺序来做的。例如，如果计算5的阶乘，递归的解决方案会以如下顺序相乘：

```
1 * 2 * 3 * 4 * 5
```

而迭代的解决方案则以相反的顺序做的乘法：

```
5 * 4 * 3 * 2 * 1
```

在这个例子中，递归和迭代两种解决方案都比较好找（虽然两者的结构差别较大）。通过重新考虑算法的结构，我们可以写出阶乘函数的一个非常简单的循环实现。但在某些情况下，想出循环版本的解决方案可能比本例要难得多。选择使用递归与否，将取决于发现迭代算法的难易程度。在阶乘的例子中，这并不太难，但在某些情况下，可能非常困难。这种情况常会遇到。

16.4 栈

是时候来了解一下函数调用是如何进行的了。一旦了解了函数调用的工作方式，会更有利于你理解递归，并且对于理解为什么有些算法很容易写出递归版本，却很难写出循环版本，会有直观的感受。

函数所使用的所有内部信息都存储在栈中。想象一叠盘子，你可以把新盘子放在这叠盘子的顶部，也可以从顶部取走盘子。栈的工作原理与一叠盘子类似，不同的是，栈中存放的不是“盘子”，而是称为栈帧的东西。当一个函数被调用时，它在栈的顶部得到一块新的栈帧，并使用这块栈帧来存储所有它要使用到的局部变量。当这个函数调用了另外一个函数时，原始的栈帧空间被保留，新的栈帧被添加到栈顶，用于为新近调用的函数存放自己的变量。当前正在执行的函数总是使用栈顶的栈帧。

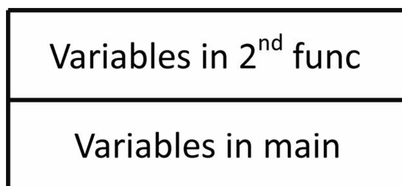
在最简单的情况下，只有main函数在执行，这时候的栈看起来像这样：

Variables in main

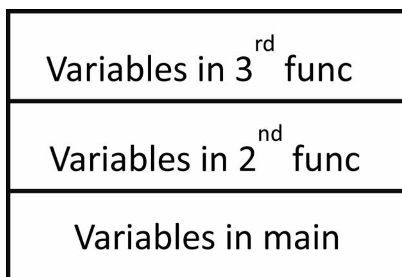
当前只有一个函数正在执行，即main函数；栈中只有main函数的变量。

现在，如果main函数调用了其他函数，那么新的函数将在main函数的顶部创建一个新的栈

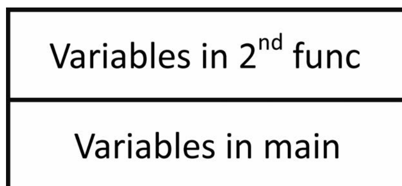
帧。看起来像这样：



当前的函数在自己的栈帧中存储自己的变量，不会干预到main函数所使用的变量。如果第二个函数调用第三个函数，这时候的栈看起来就像这样：



每个新调用的函数都有自己的栈帧；每次函数调用都会创建新的栈帧。一旦函数返回，栈将回到函数调用之前的样子：



如果第二个函数返回到main函数，此时栈回到了只有一个栈帧的样子：



当前正在执行的函数的栈帧处于活动状态，它始终在栈的顶部。

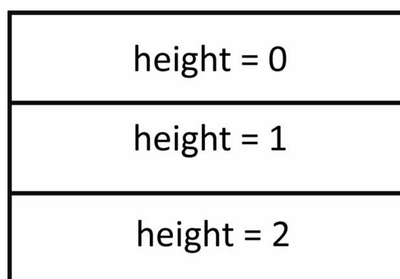
除了保存函数所用到的变量，栈帧中还包含了传递到函数的参数，以及函数结束时应该返回到的函数的代码行。换句话说，栈帧中存储着函数从何处被调用的信息，以及函数使用的所有数据。递归调用一个函数将为新调用的函数创建新的栈帧，即使是相同的函数也是如此。这就是递归能正确工作的原因：每个函数调用都有自己独立的栈帧，包括自己的参数和变量。这使得每个函数调用拥有自己的信息，因此每个函数都可以把原问题的更小版本表示成自己的变量来处理。

正如示意图中所看到的，当函数结束后，该函数从栈顶中删除自己的栈帧，并返回到其调用函数的执行点。通过删除该函数的栈帧，恢复出其调用函数所使用的栈帧。

关键在于：栈帧中保存着函数的返回位置，并且，函数执行结束后会从栈中删除其栈帧。栈帧如果不正确，函数返回后，调用函数将不能继续正确执行——比如，调用函数的局部变量没有获得返回值。

不妨这样想：当新函数被调用时，之前的函数会把继续执行所需要的一切数据保留下来。这就好比当你在做一个项目时，中途决定去吃饭，就会做个记号，标明项目做到哪儿了，以便饭后继续工作。栈允许计算机在任何时刻对当前的运行信息做极其详细的记录。

这里有个栈，它演示了对buildWall函数的三次递归调用，从高度为2时开始执行。可以看到，每个栈帧都保留传递到buildWall函数中的高度值。（注意，当调用高度值为0的buildWall函数时，栈停止了增长，因为刚好到地面了。）



这种绘制栈的方法也常简写成这样：

```
buildWall( x = 0 )
buildWall( x = 1 )
buildWall( height = 2 )
main( )
```

每个函数都显示在其调用函数的顶部，函数的参数同时也显示出来。使用这种表示方法，可以帮助你理解一个特定的递归函数是如何工作的。有时你会发现，除了显示函数名和函数参数外，在每个栈帧的旁边写上局部变量也很有用。

16.4.1　栈的力量

从递归中能够获得用于后续计算的关键值的主要原因是：在递归中你拥有的是一堆函数调用构成的栈，而不仅仅是一个栈帧。递归算法可以利用存储在每个栈帧中的所有的额外信息，而循环只能获得一个栈帧里的一组局部变量。因此，递归函数可以等待递归调用返回，从而接收到需要的值，然后从停止点继续执行。要想写一个以这种方式工作的循环，你需要自己实现一个栈。

16.4.2　递归的缺点

栈的大小是固定的，这也就意味着不能有无限的递归。递归到某些时候，栈顶将会没有更多空间来添加新的栈帧——就好像橱柜的空间被挤满，不能再增加一个盘子一样。

下面是一个简单的例子，理论上这个递归可以无限执行：

```
void recurse ()
{
    recurse(); // 函数调用自身
}

int main ()
{
    recurse(); // 开始递归
}
```

但最终，栈空间会消耗一空，程序将因栈溢出而崩溃。当栈空间不足时，就会发生栈溢出。这时，没有任何空间来做函数调用，如果你的程序尝试此操作就会崩溃。这种类型的崩溃比较罕见，通常是递归函数的基线条件没写好才导致的结果。例如，前面我写的阶乘函数的例子就有一个小问题：它在基线条件中没有对负数进行检查。如果调用函数传入-1，这几乎肯定会发生栈溢出。（试试看，别担心！栈溢出会导致程序崩溃，但不会损坏你的电脑。）

下面是一个简单的程序，向你展示一个很小的函数要递归调用多少次才会耗光栈空间。（函数的栈帧越大，能进行的递归调用次数就越少。不过，如果基线条件编写正确，这种极限情况基本不会发生。）

```
#include <iostream>

using namespace std;

void recurse (int count) //每次调用的count都不一样
{
    cout << count << "\n";
    // 这里没有必要写一条语句来专门递增count
    // 因为每个函数的变量是独立的
    // (因此每个栈帧中的count将被初始化为上一个函数的count加1)
    recurse( count + 1 );
}

int main ()
{
    recurse( 1 ); //第一次函数调用，因此置为1
}
```

16.4.3　调试栈溢出

当你尝试调试栈溢出时，最重要的是找出是哪个函数（或哪组函数）在不断地增加栈帧。例

如，如果使用的是调试器（将在第20章讲到），当程序最终崩溃时，你会看到上述例子中栈变成了类似如下的样子：

```
recurse( 10000 );
recurse( 9999 );
recurse( 9998 );
...
recurse( 1 )
main()
```

这种情况很容易分析，因为只有一个函数被调用。显然，这个函数缺失了某种基线条件，导致递归参数达到一定大小而递归调用却没有停止。

有时候，会出现两个函数*相互递归*调用的情况。

再次以阶乘为例。这是一个人为的例子，我们使用两个函数来计算阶乘：一个用于计算偶数的，一个计算奇数的：

```
int factorial_odd (int x)
{
    if ( x == 0 )
    {
        return 1;
    }
    return factorial_even( x - 1 );
}

int factorial_even (int x)
{
    if ( x == 0 )
    {
        return 1;
    }
    return factorial_odd( x - 1 );
}

int factorial (int x)
{
    if ( x % 2 == 0 )
    {
        return factorial_even( x );
    }
    else
    {
        return factorial_odd( x );
    }
}
```

请注意，这里的基线条件没有提防负数输入。调用`factorial(-1)`会导致这样的调用栈：

```
factorial_even( -10000 )
```

```
factorial_odd( -9999 )
factorial_even( -9998 )
factorial_odd( -9997 )
```

仅通过查看栈，我们就能知道基线条件存在一个问题，这两个函数一直在相互调用。下一步是查看代码，尝试找出哪个函数应该在基线条件中对负数进行检查。对于计算阶乘的情况，合理的方式是两个函数都应该分别包含对负数的检查；而在其他情况下，可能只有一个函数负责检查最后的基线条件。

当你调试复杂的递归调用时，通过查看栈有助于发现不断重复的一系列函数——在这个例子中，只有factorial_even和factorial_odd。但在某些情况下，重复的函数调用之间的时间间隔可能非常长。你必须找到整个重复的函数集合，然后揪出这些函数重复调用的原因。

16.4.4　性能

递归需要做许多函数调用，每个函数调用都需要设置一个栈帧，并传递参数，这些都增加了时间开销，而这些开销在循环中并没有。在绝大多数情况下，现代计算机中这些开销的影响并不显著。但如果你的代码会频繁地执行（比如短时间内执行百万次甚至上亿次），你就必须关注函数调用的性能问题了。

16.5　盘点递归

递归使得我们能够创建出这类算法：将问题分解成更小版本的相同问题，从而解决原问题。递归比循环更强大的地方还在于，递归函数维持着一个保存每次递归调用当前状态的栈，允许函数获得子问题的结果后继续处理。

算法的递归实现通常比等效的循环实现更加自然。下一章，我们会看到更多这样的例子，涵盖了二叉树的内容。当你开发更多的代码时，会发现使用递归比仅使用循环，更容易考虑更大范围的问题。

下面是一些何时使用递归或循环的经验法则。

适合用递归的情况：

(1) 问题的解决需要将问题分解成相同问题的较小版本，且存在一个明显能用循环来实现的方案；
(2) 你正在处理的数据结构是递归的（比如链表）。

适合用循环的情况：

(1) 很明显能用一个简单的循环来解决问题（例如，要将一串数字相加，你当然可以写一个

递归函数，但这不值得）；

(2) 正在处理的数据结构使用数字进行索引时，如数组。

16.6 问答题

(1) 下列哪种情况是尾递归？

A. 当你呼唤自己的狗时
B. 当一个函数调用自身时
C. 当一个递归函数所作的最后一件事是调用自身时
D. 当你可以将一个递归算法改写成循环算法时

(2) 何种情况下适合使用递归？

A. 当你不能使用循环来实现算法时
B. 当从子问题角度要比从循环角度能更自然地表达一个算法时
C. 永远不要，真的，它太难了
D. 在使用数组和链表时

(3) 一个递归算法需要满足什么要素？

A. 基线条件和递归调用
B. 基线条件和将问题分解成问题本身的更小版本的方式
C. 重组问题的较小版本的方式
D. 以上皆是

(4) 当基线条件不完整时，会发生什么？

A. 该算法可能提前完成
B. 编译器将检测到这个问题并报错
C. 这是没有问题的
D. 可能会发生栈溢出

16.7 实践题

(1) 写一个递归函数来计算幂函数：`pow(x, y) = x^y`。

(2) 写一个递归函数，它接受一个数组，并以相反的顺序显示出数组元素，不能从数组末尾开始扫描索引。（换句话说，不要写一个等效的循环，它从数组末尾开始输出数组元素。）

(3) 写一个递归算法，从一个链表中删除元素；写一个递归算法，将元素添加到链表中。尝

试用迭代来实现相同的算法。递归实现和迭代实现，哪个感觉更自然呢？

(4) 写一个递归函数，它接受一个排好序的数组和一个目标元素，在数组中查找目标元素（如果元素不在数组中的话，返回目标函数的索引或 –1）。这个搜索能跑多快呢？能否不用查看每个元素就能找到目标元素？

(5) 写一个递归函数来解决汉诺塔问题。以下是一个描述了汉诺塔问题的网站，试试看吧：http://www.mazeworks.com/hanoi/index.htm。

第 17 章 二叉树

注意：本章我要介绍一个有趣而实用的基本数据结构——二叉树。二叉树是一个使用递归和指针的最完美例子，我们可以用它来做一些惊人的工作。不过，开启二叉树的学习之旅前，你需要切实理解递归和链表的基本概念。何出此言呢？我见过不止一个学生敷衍地看了下指针和链表，就匆匆进入二叉树的泥潭中苦苦挣扎。二叉树本身没什么难的，理解它并不困难，但这建立在你有一个坚实的基础之上。如果你对本章的概念理解上有困难，那就需要更深入地学习指针和递归——请重读之前几章并完成练习。

链表是一个伟大的技术，很适合进行列表操作，但在链表中查找一个特定元素可能会花费大量时间。此外，哪怕用数组来存储，如果列表中的数据特别多，想查找一个特定元素也同样非常耗时。或许你可以试试对数组进行排序，这样就能实现快速搜索。但是，在数组中插入新元素就会变得很困难——如果要保持数组的排序，那么每次插入新元素时都要移动很多的元素。另外，快速地查找到东西是很重要的，举几个例子。

(1) 如果你正在创建一个《魔兽世界》那样的MMORPG游戏（大型多人在线角色扮演游戏），要让玩家能够快速登录游戏，就必须能够迅速查找玩家。

(2) 如果你正在编写信用卡处理软件，它需要支持每小时处理数以百万计的交易，就要求能够迅速地找到一个信用卡号码的账户余额。

(3) 如果你正在给智能手机那样的低功率设备写软件，需要将地址簿显示给用户，又不希望用户因为你使用了一个缓慢的数据结构而苦苦等待。

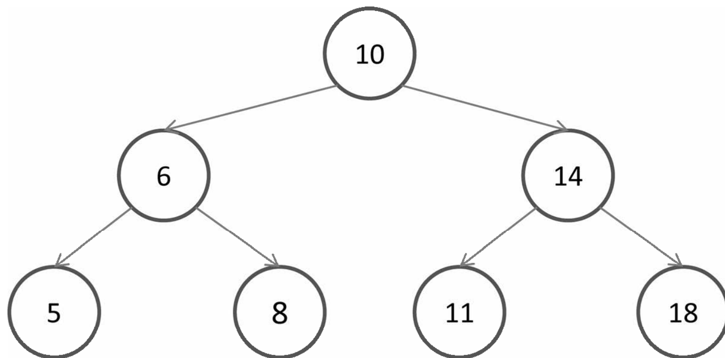
本章将会介绍解决以上问题（但不仅限于这些问题）所需要的工具。

解决此类问题的基本思路是，将元素存储在类似链表的结构中——也就是说，使用指针指向结构体类型的内存，就像我们在链表中做的一样——但要以一种更容易搜索数值的方式。为此，我们需要在内存中存储更精巧的结构化数据，而不仅仅是一个简单的列表。

来看看结构化的数据到底是什么。刚开始时，你只会使用数组，数组仅仅是一个顺序列表，没有能力来提供其他任何数据结构。链表使用指针来逐步构建一个顺序列表，但它没有利用指针所具有的灵活性来构建更精巧的数据结构。

所谓的“更精巧的数据结构”指什么呢？首先，可以构建一个数据结构，它能够同时拥有不止一个“下一个节点”。为什么要这么做呢？如果你有两个“下一个节点”，其中一个代表比当前元素小的元素，另一个代表比当前元素大的元素，这种数据结构就称为*二叉树*。之所以如此命名，是因为在二叉树中，每个节点最多有两个分支。这里的“下一个节点”称为*子节点*，指向一个子节点的节点称为该子节点的*父节点*。

一棵二叉树如下所示：



注意，在这棵树中，每个元素的左子节点都是一个比该元素小的值，而每个元素的右子节点比该元素大。节点10是整棵树的父节点。它的两个子节点，节点6和节点14，分别是自己衍生出的小二叉树的父节点。这些小二叉树称为*子树*。

二叉树的一个重要特性是，一个节点的每个子节点本身就是一棵完整的二叉树。这一特征，结合上“左子节点比当前节点小，右子节点比当前节点大”这一规则，使得寻找一棵树中的某个节点的算法设计起来很容易。首先，查看当前节点的值，如果它等于搜索目标，则搜索结束，大功告成；如果搜索目标小于当前节点的值，你往左边的树中找；否则，到右边的树去找。这个算法能够有效，主要因为左子树中的每个节点都小于当前节点，而右子树中的每个节点都大于当前节点。

最理想的二叉树是*平衡树*，即左子树与右子树的节点数量相同。对于一棵平衡树来说，每个子树是整棵树的一半大小，如果你正在查找树中的某个值，每到一个子节点，你的搜索就可以排除掉一半的元素。所以，如果有一棵1000个元素的平衡树，你可以立即砍掉500个元素。搜索就减少到在一棵500个元素的子树中进行。对一棵500个元素的树进行搜索，我们再次可以砍掉大约一半的元素，约250个。继续这样每到一个节点就排除掉一半的元素，不用多久就能找到想要找的元素。总共需要多少次拆分树的操作才能到达只有一个节点的树呢？答案是$\log_2 n$，其中n为整棵树的节点数量。这个值很小，即使对于非常大的树（对于一棵约有40亿个元素的树，$\log_2 n$为32，这意味着，其搜索速度比对同等大小的链表进行同样的搜索要快近1亿倍，因为在链表中你必须要逐个地查看每个元素）。然而，如果这棵树不平衡，可能就不能每次砍去树的一半元素。在最

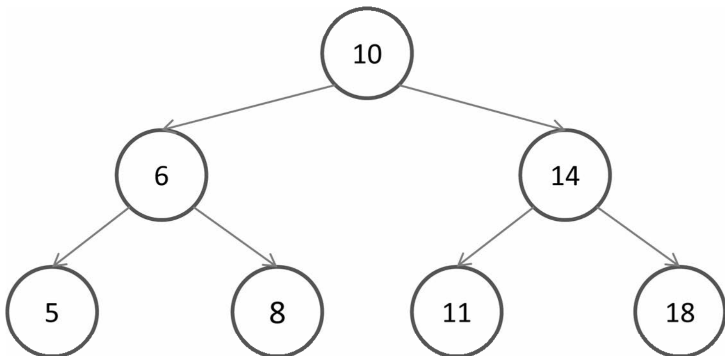
坏情况下，每个节点只有一个子节点，也就是说这棵树本质上是一个链表，只是比普通的链表多了一些额外的指针，那么其搜索过程就会退化到要遍历全部的n个元素。

如你所见，当一棵树大致平衡时（没有必要一定要完全平衡），搜索节点的速度要远远快于在链表中的搜索。这一切归根结底是因为我们可以根据自己的喜好来结构化内存，而不是止步于简单的列表[①]。

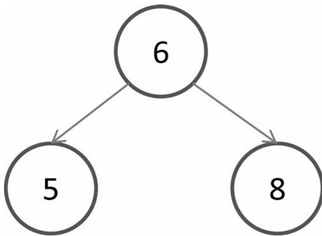
谈谈“树”

为了便于理解二叉树的示例代码，我们需要一种简便的方式来指代“树”的不同部分，因此要建立一些基本的规约来绘制和指代一棵“树”。

最基本的“树”是空树，用NULL来表示。当我画“树”的示意图时，将不画出到空树的链接。每当想提及一个特定的子树，我会说“<头为[父节点的值]的树>”。例如，在这棵树中：



<头为6的树>指代的就是这个子树：



① 此处讨论的基本二叉树，只有极少数情况下会最终与链表结构相同，这取决于节点的插入顺序。还有更复杂的二叉树类型，它们总是迫使树适当的平衡，此种数据结构称为红黑树，但这超出了本书的讲述范围：http://en.wikipedia.org/wiki/Red%E2%80%93black_tree。

实现二叉树

来看看简单实现一个二叉树所需的代码。首先，我们声明一个节点结构体：

```
struct node
{
    int key_value;
    node *p_left;
    node *p_right;
};
```

我们的节点可以将key_value的值作为一个简单的整数值存储下来，并且包含两个子树，分别是p_left和p_right。

这几个是你会在二叉树上执行的常用函数：插入节点到树中，搜索树中的某个值，从树中删除某个节点，删除整棵树以释放内存。

```
node* insert (node* p_tree, int key);
node *search (node* p_tree, int key);
void destroyTree (node* p_tree);
node *remove (node* p_tree, int key);
```

在树中插入新节点

首先学习使用递归算法来实现树节点的插入。递归算法能用在树上，是因为每棵树都包含两棵更小的树，所以整个数据结构本身就是递归的。（假设每棵树都包含一个数组或是一个指向链表的指针，那么这种数据结构就不是递归的了。）

函数接受一个key值和一棵已存在的树（可能为空），返回包含此插入值的新树。

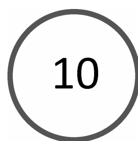
```
node* insert (node *p_tree, int key)
{
    // 基线条件：我们到达了一棵空树，需要将新节点插入到这里
    if ( p_tree == NULL )
    {
        node* p_new_tree = new node;
        p_new_tree->p_left = NULL;
        p_new_tree->p_right = NULL;
        p_new_tree->key_value = key;
        return p_new_tree;
    }
    // 决定将新节点插入到左子树或右子树中
    // 取决于新节点的值
    if( key < p_tree->key_value )
    {
        // 根据p_tree -> left和新增的key值，构建一棵新树，
        // 然后用一个指向新树的指针来替换现有的p_tree -> left
        // 之所以需要替换现有的p_tree -> left，是为了防止
        // 原有的p_tree -> left为NULL的情况（如果不为NULL，
```

```
            // p_tree->p_left实际上不会改变，但替换下也无妨)
            p_tree->p_left = insert( p_tree->p_left, key );
        }
        else
        {
            // 插入到右子树的情况与插入到左子树是对称的
            p_tree->p_right = insert( p_tree->p_right, key );

        }
        return p_tree;
    }
```

此算法的基本逻辑是：如果当前拥有的是一棵空树，那就创建一棵新的树。若非空树，那么如果要插入的值大于当前节点，就将其插入左子树中，并用新创建的子树替换原来的左子树；否则就将新节点插入右子树中，并做同样的替换。

让我们在实例中看看这段代码——将一棵空树构建成有两个节点的树。如果将值10插入一棵空树（NULL）中，立即达到了基线条件，其结果是一棵非常简单的树：



这棵树的两个子树都指向了NULL。

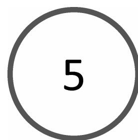
如果再将值5插入到树中，将调用函数：

```
insert( <头为10的树>, 5 )
```

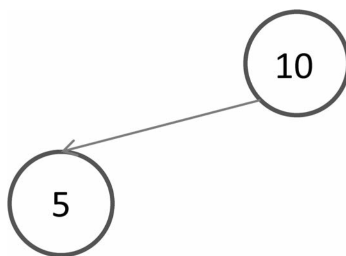
由于5比10小，我们将对左子树进行递归调用：

```
insert( NULL, 5 )
insert( <头为10的树>, 5 )
```

函数insert(NULL, 5)将创建一棵新的树，并将它返回：



当函数insert(<头为10的树>, 5)收到返回的树时，会将两棵树链接到一起。在这种情况下，头为10的树的左子树原本为NULL，被替换后就变成了一棵全新的树。



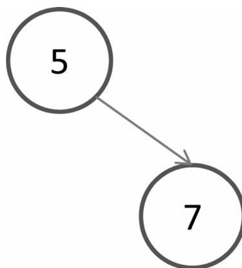
如果我们接着在树中插入7，将递归调用：

```
insert( NULL, 7 )
insert( <头为5的树>, 7 )
insert( <头为10的树>, 7 )
```

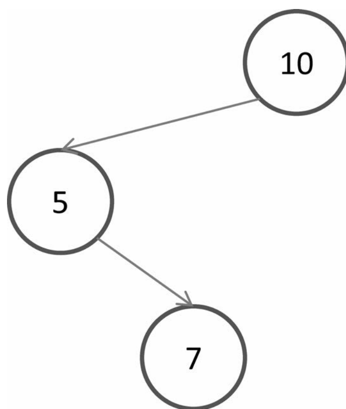
首先，insert(NULL, 7)返回一棵新树：



然后，insert(<头为5的树>, 7)将头为7的新子树链接起来，就像这样：



最后，这颗树被返回给insert(<头为10的树>, 7)，并在insert(<头为10的树>, 7)中被链接起来：



由于节点10已经有一个指针指向了节点5，重新再链接节点10的左子树到头为5的树这一步骤并不是必需的，但这么做省去了在代码中检查左子树是否为空这一额外的条件检查。

在树中搜索

现在，来看看如何实现在树中进行搜索，其基本逻辑与在树中插入新节点的算法几乎完全一样：首先，检查两个基线条件（是否发现目标节点，或是否到达了一个空树）；如果基线条件不满足，就确定应该去哪个子树中搜索。

```
node *search (node *p_tree, int key)
{
    // 如果到达了空树，很明显，值key不在这棵树中!
    if ( p_tree == NULL )
    {
        return NULL;
    }
    // 如果找到了值key，搜索完成!
    else if ( key == p_tree->key_value )
    {
        return p_tree;
    }
    // 否则，尝试在左子树或右子树中寻找
    else if ( key < p_tree->key_value )
    {
        return search( p_tree->p_left, key );
    }
    else
    {
        return search( p_tree->p_right, key );
    }
}
```

上面的search函数首先检查两个基线条件：是否到达树的分支末端或是否找到了值key。无论哪种情况，我们都知道应该返回什么：如果到达树的分支末端，就返回NULL；如果找到了key值，就返回这棵树本身。

如果基线条件不满足，我们就在子树中找key值，从而减小了问题。在左子树还是在右子树中查找，取决于key的值。请注意，每次递归调用，树的大小正如本章开头所讲——约减少了一半。在本章开头，我们还看到，在一棵平衡二叉树中搜索所花费的时间正比于$\log_2 n$，当数据量很大时，这远比通过链表或数组进行搜索要快得多。

删除树

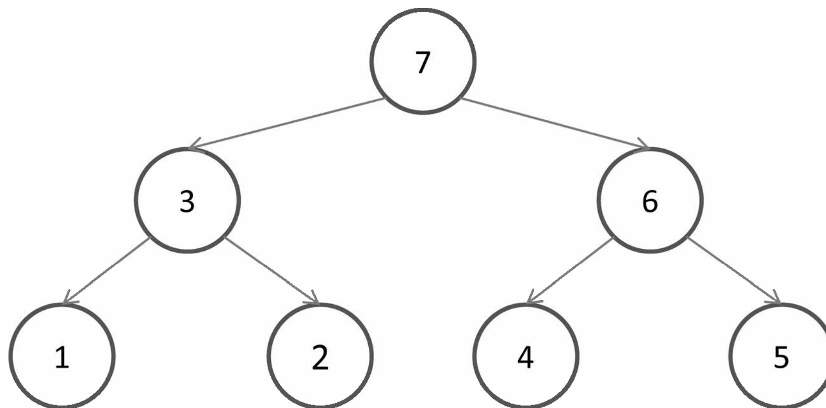
destroy_tree函数也应该是递归的。该算法将先删除当前节点的两个子树，然后再删除当前节点。

```
void destroy_tree (node *p_tree)
{
    if ( p_tree != NULL )
    {
        destroy_tree( p_tree->p_left );
        destroy_tree( p_tree->p_right );
        delete p_tree;
    }
}
```

为了帮助理解整个递归调用过程，你可以在删除节点前输出节点的值：

```
void destroy_tree (node *p_tree)
{
    if ( p_tree != NULL )
    {
        destroy_tree( p_tree->p_left );
        destroy_tree( p_tree->p_right );
        cout << "Deleting node: " << p_tree->key_value;
        delete p_tree;
    }
}
```

你会看到，那棵树是“自下而上”被删除的。节点5和节点8首先被删除，接着是节点6；然后删除树的另一边，删除节点11和节点18，接着是节点14；最后，当所有的子节点都被删除时，删除节点10。树中的值并不重要，重要的是节点的位置。我在下面的二叉树中放置的是节点删除的顺序，而不是每个节点的值：



手动运行代码时，这类型的图对于理解其删除过程相当有帮助，它使整个过程变得更加清晰。

删除树的递归算法就是一个递归算法难以用迭代实现的例子！你需要写出这样一个循环：它能以某种方式同时处理树的左支和右支！也就是说，你需要能够在删除一棵子树的同时，跟踪到要删除的第二棵子树，并且得在树的每一层都要做到这一点。栈可以用来记录你的位置。你可以

这样可视化其过程，每个栈帧都有效地存储了树的哪一个分支已经被删除：

```
destroy_tree( <子树> )
destroy_tree( <树> ) - 知道这里的“子树”是左子树还是右子树
```

每个栈帧通过函数将要继续执行的位置，来获知树的哪个部分需要删除。第一次调用destroy_tree时，栈帧告诉程序继续执行第二次。第二次调用destroy_tree时，栈帧告诉程序继续删除树。由于每次函数调用都有自己的栈帧，所以它跟踪了树被销毁的整个过程，每一层每一次它都有记录。

实现其非递归算法的唯一方式是，用一个数据结构来为我们保存相同数量的信息。例如，你可以写一个模拟栈的函数，它维护着一个链表，在链表（模拟栈）中记录正在销毁处理的子树。子树的哪一边还未删除也在链表中记录着。接着，你可以写一个循环算法，将子树添加到链表中，当子树被完全删除时，将其从列表中移除。换句话说，递归可以利用内置的栈数据结构，而不必由你自己编写。作为练习，我建议你尝试完成destroy_tree的非递归实现。你会看到，使用递归要比创建自己的栈更易于表达，从而对递归有更深的理解。

从树中删除节点

从二叉树中删除节点的算法就复杂多了。该算法的基本结构跟我们前面见过的模式差不多：如果到达一棵空树，任务结束；如果要删除的值在左子树中，到左子树中搜索并删除该值；如果在右子树中，则到右子树中搜索删除；如果找到了这个值，将其删除。

```
node* remove (node* p_tree, int key)
{
    if ( p_tree == NULL )
    {
        return NULL;
    }
    if ( p_tree->key_value == key )
    {
        // 这里该怎么办
    }
    else if ( key < p_tree-="">key_value )
    {
        p_tree->left = remove( p_tree->left, key );
    }
    else
    {
        p_tree->right = remove( p_tree->right, key );
    }
    return p_tree;
}
```

但这个看似完美的操作中隐藏的一个麻烦问题在于其中的一个基线条件。当你真正找到了要删除的值时，究竟需要做些什么呢？别忘了，二叉树要始终满足下列条件：

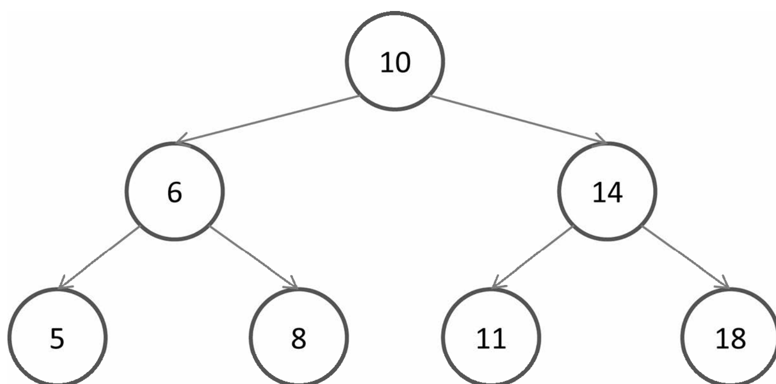
> 当前节点的左子树中的每个值都必须小于当前节点的值；当前节点的右子树中的每个值必须大于当前节点的值。

有三个基线条件需要考虑：

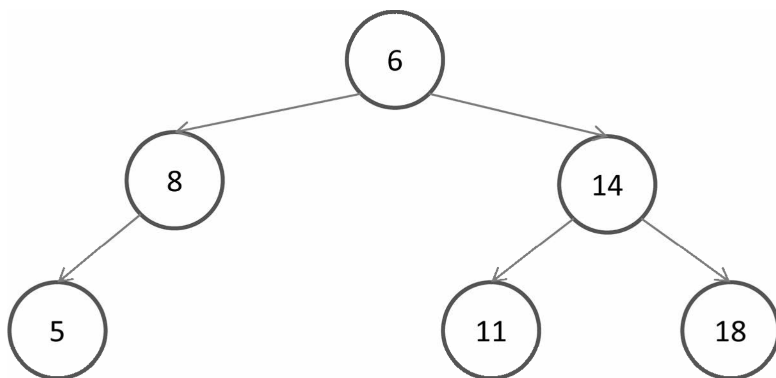
(1) 被删除的节点没有子节点；
(2) 被删除的节点只有一个子节点；
(3) 被删除的节点有两个子节点。

情况(1)最容易处理：如果要删除的节点没有任何子节点，返回`NULL`即可；情况(2)也不难：如果只有一个子节点，将该子节点返回；但是情况(3)就复杂多了。

我们不能随便选一个子节点提升上来，然后自以为万事大吉。例如，如果我们选择提升左子节点来替代要删除的元素，会怎样呢？如果这么做，该节点的右边元素会发生什么？考虑下早期时候用过的这个例子：

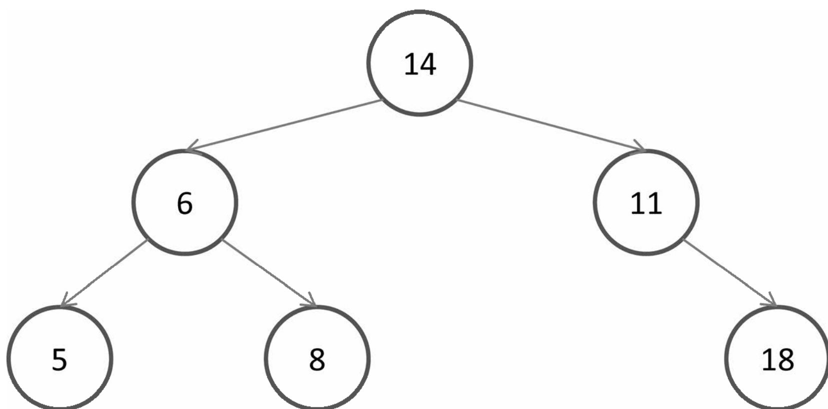


如果要删除的元素是节点`10`，该怎么办呢？不能只是提升元素`6`来替换`10`，否则，你最终得到的是这样一棵树：



现在，节点8在节点6的左边，尽管8大于6。很显然，这棵树被破坏了——在树中对值8进行搜索，将进入到节点6的右边，永远找不到节点8。

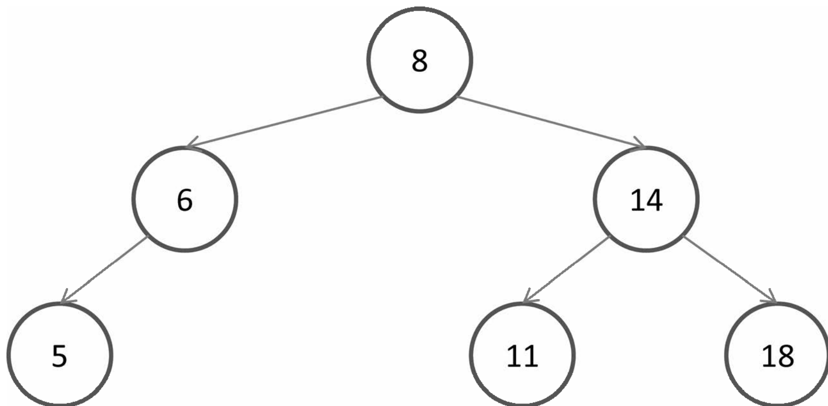
类似地，我们不能仅是提升右子节点：



这里，11比14小，但它却在树的右边，这是不应该的。在二叉树中，究竟提升哪个节点，需要慎之又慎。

所以，怎么办呢？既然一个节点的左边所有节点的值一定小于该节点的值，那么，为什么不找出要删除的节点左边的所有节点中最大的值，并把它提升到这棵树的顶端呢？由于它是这棵树左侧的最大值，用它来替换当前节点是绝对安全的：既保证了该节点比其左侧的其他节点都大，同时由于它本来就在这棵树的左侧，也肯定小于其右侧的每个节点[①]。

在刚才的例子中，由于8是节点10左侧最大的值，最终的树会是这样：



① 同理，你也可以选择这棵树右侧值最小的节点。在实践中，一个好的算法不会始终选择同一个方向，否则会产生不平衡树。但为了简便起见，我们将忽视这个随机化过程，而使用比较简单的版本。

为此，我们需要一个算法来找出一棵树左侧存储的最大值，即find_max函数。我们可以利用“更大的值总是出现在子树右侧”这一性质，来实现find_max函数。因此可以沿着一棵树的右分支往下走，直到抵达NULL为止。换句话说，对于find_max函数而言，它接受一棵树，返回这棵树中的最大值。我们可以把这棵树的所有右指针看做构成了一个链表：

```
node* find_max (node* p_tree)
{
    if ( p_tree == NULL )
    {
        return NULL;
    }
    if ( p_tree->p_right == NULL )
    {
        return p_tree;
    }
    return find_max( p_tree->p_right );
}
```

注意，这里需要两个基线条件：一个用于空树的情况，一个用于抵达右分支末端的情况[①]。为了返回指向最后一个节点的指针，我们需要趁着指针还有效时“向前看”一个节点。

让我们来看看能否用find_max完成remove函数。在基线条件中，如果find_max返回NULL，我们就知道可以只使用左侧的树来取代删除的节点，因为没有一个值比它大。否则，我们需要用find_max返回的结果来取代删除的节点。

```
node* remove (node* p_tree, int key)
{
    if ( p_tree == NULL )
    {
        return NULL;
    }
    if ( p_tree->key_value == key )
    {
        // 前两种情况处理没有任何子节点，或只有一个子节点
        if ( p_tree->p_left == NULL )
        {
            node* p_right_subtree = p_tree->p_right;
            delete p_tree;
            // 如果没有任何子节点，返回NULL
            // 这正是我们想要的
            return p_right_subtree;
        }
        if ( p_tree->p_right == NULL )
        {
            node* p_left_subtree = p_tree->p_left;
            delete p_tree;
```

① 我们实现删除节点的方式，实际上不需要做基线条件一（必须是空树）的检测，但预防糟糕的输入是一种优秀的编码风格。

```
            // 这里，总能返回一个有效的节点，
            // 因为从上一个if语句中，我们知道p_tree->p_left不是NULL
            return p_left_subtree;
        }
        node* p_max_node = find_max( p_tree->p_left );
        p_max_node->p_left = p_tree->p_left;
        p_max_node->p_right = p_tree->p_right;
        delete p_tree;
        return p_max_node;
    }
    else if ( key < p_tree->key_value )
    {
        p_tree->p_left = remove( p_tree->p_left, key );
    }
    else
    {
        P_tree->p_right = remove( p_tree->p_right, key );
    }
    return p_tree;
}
```

大功告成了吗？不，还有一个藏得很深的问题：我们并没有将max_node从原来的位置中真正删除掉！这意味着，在树中的某个地方，存在一个指向max_node的指针，又指回了树。而且，max_node原来的子树还变得不可达了。

我们需要从树中删除max_node。 幸好，我们知道max_node没有右子树，只有左子树，也就是说它只有最多一个子节点[1]。这种情况就变得简单多了。我们只需要修改max_node的父节点指向max_node的左子树。

我们可以写一个简单的函数,它接受一个指向max_node的指针和一棵包含max_node的树的头，返回一棵新的树，这棵树妥善地删除了max_node。注意，这个函数能有效运行，依赖于max_node不存在右子树！

```
node* remove_max_node (node* p_tree, node* p_max_node)
{
    // 预防性代码——实际不应该到达这里
    if ( p_tree == NULL )
    {
        return NULL;
    }
    // 找到了目标节点，现在要替换它
    if ( p_tree == p_max_node )
    {
        // 可以这么做的唯一理由是，
        // 我们知道p_max_node->p_right为NULL，
        // 所以不会丢失任何信息（节点）
        // 如果p_max_node没有左子树，则仅返回NULL，
        // 于是p_max_node将被一棵空树取代，
```

① 这点是已知的，因为它是一棵子树的最大值，所以不可能有右子节点。

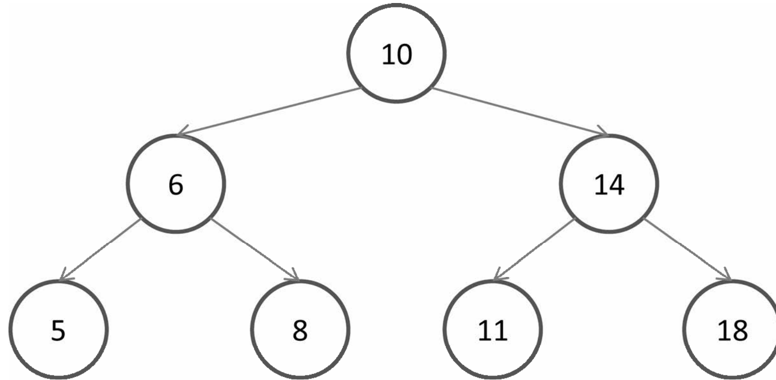
```
        // 而这正是我们想要的
        return p_max_node->p_left;
    }
    // 每次递归调用都将右子树替换成一棵不包含p_max_node的新子树
    p_tree->p_right = remove_max_node( p_tree->p_right, p_max_node );
    return p_tree;
}
```

有了这个辅助函数，现在我们可以轻松地修改remove函数，使得在用左侧的最大节点替换掉被删除节点前，先把这个最大节点从左子树中剥离掉。

```
node* remove (node* p_tree, int key)
{
    if ( p_tree == NULL )
    {
        return NULL;
    }
    if ( p_tree->key_value == key )
    {
        // 前两种情况处理没有任何子节点，或只有一个子节点
        if ( p_tree->p_left == NULL )
        {
            node* p_right_subtree = p_tree->p_right;
            delete p_tree;
            // 如果没有任何子节点，返回NULL
            // 这正是我们想要的
            return p_right_subtree;
        }
        if ( p_tree->p_right == NULL )
        {
            node* p_left_subtree = p_tree->p_left;
            delete p_tree;
            // 这里，总能返回一个有效的节点，
            // 因为从上一个if语句中，我们知道p_tree->p_left不是NULL
            return p_left_subtree;
        }
        node* p_max_node = find_max( p_tree->p_left );
        // 由于p_max_node来自左子树，
        // 我们需要在把该左子树重链接回树之前将p_max_node节点删除
        p_max_node->p_left =
            remove_max_node( p_tree->p_left, p_max_node );
        p_max_node->p_right = p_tree->p_right;
        delete p_tree;
        return p_max_node;
    }
    else if ( key < p_tree->key_value )
    {
        p_tree->p_left = remove( p_tree->p_left, key );
    }
    else
    {
        p_tree->p_right = remove( p_tree->p_right, key );
```

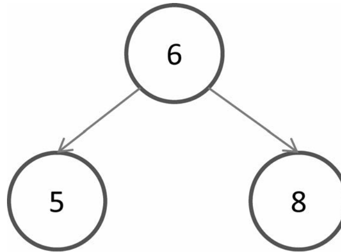
```
    }
    return p_tree;
}
```

来看看这段代码在之前作为例子的树中是如何执行的：

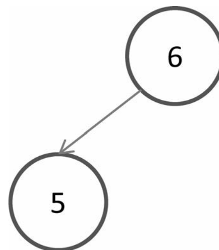


如果我们打算从树中删除节点10，remove函数将立即到达“找到目标节点”这一基线条件。它会发现节点10既有左子树也有右子树，因此，它会到头为6的子树中找到其中值最大的节点——节点8，然后将节点8的左指针指向头为6的新树，这棵新树不包含节点8。

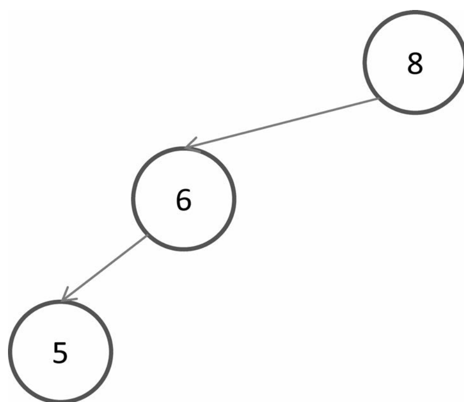
从子树中很容易删除节点8。我们从这棵子树开始说起：



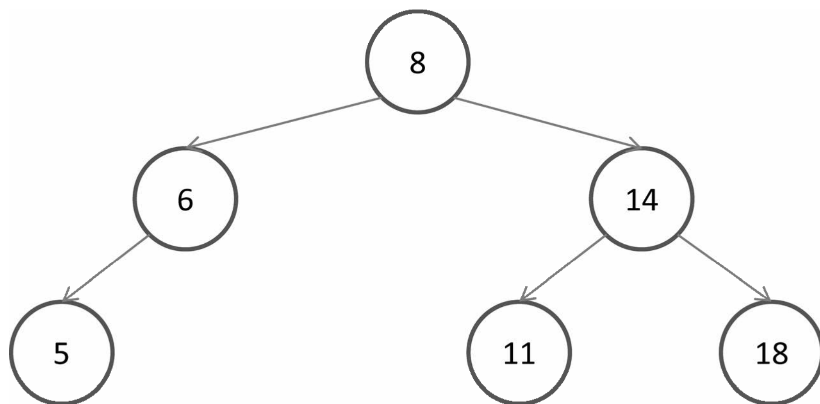
第一次调用remove_max_node函数，发现节点6不是想要删除的节点，因此，对头为8的子树再次递归调用remove_max_node函数。由于节点8正是想找的节点，我们返回节点8的左子树（NULL），从而节点6的右指针更改成了指向NULL。这棵子树现在变成这样：



在remove函数的调用中，我们现在收到了从remove_max_node返回的树（如上所示），并将节点8的左指针设置成指向这棵树。现在，新树成了这样：



最后，节点8的右指针被设置成指向头为14的右子树。此刻，树彻底重建好了：



于是我们释放原来的节点10所占用的空间。

你可以从本章中找到全部的源代码。在文件binary_tree.cpp中，还有一个简单的程序包含了树的各种操作。

17.1　在现实世界中使用二叉树

尽管我已经谈了很多与快速搜索相关的知识，但你可能还会犯嘀咕：从一个数据结构中找到一个特定值的速度有多快真的很重要吗？电脑速度不是已经够快了吗？我究竟什么时候需要用到快速搜索呢？

通常在两种情况下，搜索速度至关重要。第一种情况是检查是否存在一个特定的值。例如，

如果你有一个游戏，它允许用户注册用户名，那就需要能够检查当前用户注册的用户名是否已被占用。如果这个游戏是魔兽世界那样的大型游戏，就要求即使有高达百万计的用户，检查速度也得非常快。由于用户名实际上是字符串，而不是整型数，因此对用户名的检查占用的时间更长，因为你需要对比每个字母。如果这种检查只做几次，你不会觉得用时很长；但如果总共要做超过百万次的比较，这个速度将慢到无法忍受。因此，使用二叉树来存储用户名肯定会使注册体验更好。如果你希望用户玩你的游戏，肯定得让注册快捷一些。

另一种情况是，你有一些与所存储的值相关联的额外数据。这种数据结构称为映射（map）。一个map中存储了一个键（key）和一个与之相关联的值（value，这个值不一定是单一的数据值，它可以是一个结构体，甚至你需要存储很多信息的话，它也可以是列表或另一个映射）。

以魔兽世界这样的游戏为例。任何大型多人在线游戏都需要一个从用户名到其密码的一个映射[①]，来处理用户登录或加载角色状态。你每次以用户名和密码登录时，魔兽世界都会到map中查找你的用户名和对应的密码，比对用户输入的密码是否有效，若有效则检索其他角色信息，让用户进入游戏。

我们可以使用二叉树来实现这样一个map。在这一过程中，可以使用键作为二叉树的插入节点（此例中为用户名），在同一节点中存储它的值（此例中为密码）。

map的概念在生活中随处可见。举一个范围更大的例子，比如信用卡公司也会用到某种形式的map。你每次使用信用卡购物，你账户里就有一些信息需要更改。成千上万的人使用信用卡，如果对每一笔信用卡交易都要遍历一次全部信用卡号码，整个世界的商业将陷入瘫痪。因此，对于给定的信用卡号码，我们必须能够快速地查找到其账户余额。要做到这一点，同样可以使用二叉树来构建每个信用卡号码到对应的账户余额之间的map。这样的话，每一笔信用卡交易就是一次简单的二叉树节点搜索。找到之后再更新存储在该节点中的余额。

如果总共有100万个信用卡号码，用二叉树来存储它们的话，这个查询平均要查看$\log_2 1\,000\,000$个节点，相当于大约20个节点。这比线性扫描节点列表效率要高5万倍。毫无疑问，信用卡公司会使用比二叉树更复杂的数据结构来处理这些问题，至少有一点：所有的账号信息需要永久保存在数据库中，而不能只是暂存在内存里。会有比简单的map更精巧，更复杂的数据结构来完成这项任务，但重要的是二叉树的思想，以及映射的构建可以用于构建更复杂的结构。

最后，甚至在一个较小范围内查找，速度也很重要。例如，手机一般具有显示来电姓名的功能。这是另一个快速查找的例子，你希望能够根据数字（在这个例子中是电话号码）迅速查找到姓名。我不知道这在手机上实际是如何实现的。地址簿可能没有大到足以利用二叉树的优势，但

① 在实践中，密码本身不会存储在map中，而是存储为散列版本。散列是一种算法，以某种方式将一个文本字符串转换成另一个文本字符串（或数字），使原来的值不可恢复。在这种情况下，密码的散列版本使我们根本不可能得到原密码。以散列形式存储密码，能够防止密码被通过盗看存储密码的文件或数据库泄露。密码的散列算法保证了两个密码极不可能存储成相同的字符串。

是你可能会想到利用map的概念来组织这些数据，而map往往以二叉树结构来建立，以允许快速查找[①]。

构建二叉树和map的代价

构建二叉树和映射有一定的时间开销。你必须将每个节点添加到树中，添加一个节点平均需要$\log_2 n$次操作（跟搜索节点一样，因为添加和搜索每次都是把树砍掉一半）。这意味着，构建整棵树实际上需要$n\log_2 n$次操作。由于对链表的每次线性搜索平均需要大约$n/2$次操作，如果这样的链表搜索做$2\log_2 n$次，所花费的时间就与构建一棵二叉树的时间相同。（何以见得呢？因为做链表搜索的总时间等于每次搜索平w均花费的时间乘以搜索的次数：$(n/2)*2\log_2 n = n\log_2 n$）。 换句话说，当你仅进行一次搜索时，没必要构建一棵二叉树；但是如果要进行多次搜索，就用二叉树吧。（一个有100万节点的映射，即使只进行大约40次查找，用二叉树也能提高平均性能。）对于要处理数百万笔交易的信用卡公司而言，答案更是显而易见。对于一部手机，这取决于你有多少电话以及地址簿的大小。（试着做些数学计算，来看看手机是否值得用二叉树。）

17.2　问答题

(1) 二叉树的主要优点是？

A. 使用指针　　B. 可以存储任意数量的数据
C. 允许数据的快速查找　　D. 从二叉树中删除节点很容易

(2) 什么情况下适合使用链表而不是二叉树？

A. 当你需要以某种方式存储数据，使得它可以快速查找时
B. 当你希望能访问排好序的数据元素时
C. 当你需要能够快速地将数据添加到前端或末端，但从不访问中间的元素时
D. 当你不需要释放正在使用的内存时

(3) 以下哪一项表述正确？

A. 数据添加到二叉树的顺序不同可以影响到最终树的结构
B. 应该排好序后再将数据插入到二叉树中，以便获得最佳的树结构
C. 如果元素是随机插入到二叉树中的，那么，链表查找节点的速度会比二叉树快
D. 二叉树永远不可能退化到跟链表相同的结构

---

① 还有其他的数据结构，比如散列表（http://en.wikipedia.org/wiki/Hash_table），也可以用来实现map。

(4) 以下关于二叉树查找节点速度快的解释，哪一项是正确的？

A. 速度一点都不快，每个节点有两个指针意味着你必须做更多的事来遍历树
B. 每经过树的一层，你大约砍掉了剩余节点数量的一半
C. 二叉树并不是真的比链表好
D. 二叉树的递归调用比链表的循环遍历要快

17.3 实践题

(1) 写一个程序，显示二叉树的内容。你能写一个程序，将二叉树的节点按排序顺序输出吗？按反向顺序输出呢？

(2) 写一个程序，计算二叉树的节点数。

(3) 写一个程序，能够检查一棵二叉树是否平衡。

(4) 写一个程序，它能检查一棵二叉树是否正确排序，即：对于一个给定的节点，是否其左侧节点都小于该节点的值，其右侧的节点都大于该节点的值。

(5) 写一个程序，不使用递归删除掉二叉树的所有节点。

(6) 实现一个简单的映射，它以二叉树形式保存地址簿。该映射的键值应该是联系人的姓名，映射的值是联系人的邮箱地址。可以在映射中添加、删除，或修改邮箱地址，当然，也应该能查找邮箱地址。程序关闭时，应该能够清除地址簿。提醒：可以使用任何标准的C++比较操作符（比如==、<或>）来比较两个字符串。

第 18 章 标准模板库

能够写自己的数据结构简直是太棒了。不过，后面几章你会发现，我们很少亲自写数据结构。别担心，我不会让你白用功的。你现在学会了很多如何在需要时构建自己的数据结构的知识，了解了几种常见的数据结构的特点。有时候，构建自己的数据结构是很有必要的。

C++较之C语言强大的功能之一是，C++编译器自带了大量的可复用代码库，我们称为标准模板库（Standard Template Library，STL）。标准模板库是一套常用的数据结构的集合，包括链表和一些基于二叉树的数据结构。这些数据结构允许你在创建时指定它们的数据类型，所以可以使用它们来存储任何类型的数据——整型、字符串、或结构体等都可以。

因为这种灵活性，在很多情况下我们可以不用为了完成基本的编程需求构建自己的数据结构，而是用标准模板库来代替。STL可以在几个重要方面提高你的代码层次：

(1) 你可以开始从需要的数据结构角度来思考问题，而不必担心自己想要的数据结构能否实现；
(2) 你可以随时使用这些顶级的数据结构，对大多数问题而言，其性能都非常好，所占空间也很少；
(3) 你不用担心所使用的数据结构进行内存分配和释放等操作的细节。

不过，使用标准模板库也有一些代价：

(1) 你需要了解标准模板库的各种接口，并学习如何使用它们；
(2) 错误使用标准模板库所造成的编译错误理解起来不是很容易；
(3) 并不是每一个你想要的数据结构标准模板库中都有。

标准模板库是一个很大的话题——有些书专讲STL[①]，所以我的讲述不可能面面俱到。本章的目的是向你介绍一下标准模板库中最常用的数据结构。在这以后，我会在适当的时候使用这些数据结构。

---

① 想深入了解STL，这本书是不错的选择：*Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*（http://www.amazon.com/Effective-STL-Specific-Standard-Template/dp/0201749629），作者为Scott Meyers。

18.1 vector，大小可变的数组

在STL中有一个称为vector的数据结构，可以用来代替数组。vector跟数组非常相似，只不过vector的大小可以自动调整，不需要编程人员关心内存分配和已存在元素的移动等细节问题。

使用vector的语法和使用数组的语法不一样。以下是声明一个数组和声明一个vector的对比：

```
int an_array[ 10 ];
```

与：

```
#include <vector>

using namespace std;
vector<int> a_vector( 10 );
```

首先，你需要包含（include）头文件vector，以便能随时使用vector数据结构。你还需要使用命名空间（namespace）std，因为vector跟cin和cout类似，都是标准库的一部分。

其次，当你声明一个vector时，必须在尖括号中标识出想要在vector中存储的数据类型：

```
vector<int>
```

这个语法使用了C++的一个特性——模板（故名标准模板库）。vector的实现方式允许它存储任何类型的数据，只要告诉编译器，该vector将存储哪种类型的数据即可。换句话说，这里实际上涉及两种类型：一种是所使用的数据结构的类型，它决定了数据的组织方式，另一种是存储在该数据结构中的数据的类型。模板可以组合不同类型的数据结构与存储在该数据结构中的不同的数据类型。

最后，vector的大小放在圆括号中，而不是方括号：

```
vector<int> a_vector( 10 );
```

我们使用到这个语法来初始化某个类型的变量。在此例中，我们将值10传给一个初始化例程，称为构造函数，该构造函数将构建一个大小为10的vector。接下来的几章中，我们将了解更多关于构造函数和对象的知识。

一旦构建好了自己的vector，你便能用和访问数组的同样方式来访问vector中的每个元素了：

```
for ( int i = 0; i < 10; i++ )
{
    a_vector[ i ] = 0;
    an_array[ i ] = 0;
}
```

18.1.1 vector的方法调用

vector中提供的功能比数组要多得多。你可以做诸如在vector的末尾添加新元素这样的事情，vector提供了执行这些操作的函数。使用这些函数的语法和你以前所见过的不同。vector利用了C++的一个特性，叫做*方法*（method），它是一个随着变量类型（在此例中，这个变量类型为vector）一起声明的函数。调用一个方法要使用新的语法，如下：

```
a_vector.size();
```

这段代码调用了a_vector的方法size，返回该vector的大小。这有点像访问一个结构体的域，所不同的是，你访问的不是域，而是该结构的方法。尽管size方法显然要对a_vector做一些操作，但你并不需要将a_vector作为一个参数传递给size方法。方法的语法知道要将a_vector作为一个隐含的参数传给size方法。

你可以看做这样的语法：

```
<variable>.<function call>( <args> );
```

就好像调用一个属于variable类型的函数一样。换句话说，它有点像写成这样：

```
<function call>( <variable>, <args> );
```

本例中，

```
a_vector.size();
```

就像是：

```
size( a_vector );
```

接下来的几章会继续介绍方法，以及如何声明和使用它们。现在你只需要知道，在vector中有很多方法可以调用，并且调用它们需要使用特殊的语法。这个特殊的语法是进行这种函数调用的唯一方式——你不能写成size(a_vector)。

18.1.2 vector的其他功能

vector还有哪些强大的功能呢？vector可以很容易地增加它所存储的值的数目，无需做任何烦琐的内存分配操作。例如，你若想添加更多的元素到vector中，可以这样写：

```
a_vector.push_back( 10 );
```

这个语句增加一个新元素到vector中。具体来说，它指的是，“添加元素10到当前vector的末尾”。vector本身会处理所有的调整大小操作。要是在数组中做这件事，你就必须分配新内存，将所有的值复制过去，最后再添加上你的新元素。当然，vector内部也要分配内存和复制元素，但

它会选择一种聪明的大小分配方式，使得如果你不断地添加新元素的话，vector不会每次都重新调整内存大小。

提醒一句：尽管你可以使用push_back添加新元素到vector的末尾，但不能简单地使用方括号来获得相同的效果。这是语言定义的一个怪癖：方括号只能用来处理已经分配的内存。究其原因可能是为了避免用户代码在无意识下进行内存分配。

因此，像这样的代码：

```
vector<int> a_vector( 10 );
a_vector[ 10 ] = 10; // 最后一个有效元素是9
```

实际上其效果不会实现，反倒可能会使程序崩溃，是相当危险的。然而，这样写：

```
vector<int> a_vector( 10 );
a_vector.push_back( 10 ); // 增加新元素到vector中
```

vector的大小会重新调整，成为11。

18.2 map

我们已经初步介绍了一下map的概念——根据一个值来找到另一个值。这种例子在编程中随处可见：实现一个可以按名称查找邮箱地址的电子邮件地址簿，通过账号查找账户信息，或是允许用户登录游戏，等等。

STL提供了非常方便的map类型，允许指定键（key）和值（value）的类型。例如，一个用来保存简单的电子邮件地址簿的数据结构，类似于你在上一章练习中做过的，可以这样来实现：

```
#include <map>
#include <string>

using namespace std;

map<string, string> name_to_email;
```

这里，我们需要告诉map数据结构两个不同的类型：第一个类型string，指的是键的类型；第二个类型也是string，指的是值的类型，本例中指邮箱地址。

STL的map有一个很大的特点是，你可以使用跟数组相同的语法，来真正的使用map。

添加一个值到map中的语法跟数组一样，所不同的是，map的键除数字外还可以是其他类型：

```
name_to_email[ "Alex Allain" ] = "webmaster@cprogramming.com";
```

访问map中的值的语法几乎完全一样：

```
cout << name_to_email[ "Alex Allain" ];
```

真是太方便了！跟使用数组一样简单，却可以存储任何类型的数据。更妙的是，与vector不同，你甚至不需要在使用[]操作符来添加元素之前，先设置map的大小。

你还可以很轻松地从map中删除元素。

如果不想再给我发邮件了，就可以用erase方法把我从你的地址簿中删除：

```
name_to_email.erase( "Alex Allain" );
```

再见！

你也可以用size方法来查看map的大小：

```
name_to_address.size();
```

还可以用empty方法来检查一个map是否为空：

```
if ( name_to_address.empty() )
{
    cout << "You have an empty address book. Don't you wish you hadn't deleted Alex?";
}
```

使用clear方法可以将map真正清除，这太直观了，你肯定不会弄错：

```
name_to_address.clear();
```

顺便说一下，STL容器使用一致的命名约定，因此你也可以在vector上使用clear、empty和size方法，跟在map上使用的方式一样。

18.3 迭代器

除了存储数据和访问单个元素，有时你可能只是希望遍历某个特定的数据结构中的每个元素。对于数组或vector容器，你可以利用数组的长度来读取每个单独的元素。但是，对于map容器，该怎么办呢？由于map里的键常常不是数字，所以我们不能总是通过一个计数器变量来遍历map中的所有键值。

STL有一个称为迭代器（iterator）的变量专门解决上述问题。迭代器允许你顺次访问任何给定的数据结构中的每个元素，即使该数据结构并未提供做这件事的简单方法。我们先来看看怎样使用一个vector的迭代器，然后再学习如何使用一个迭代器来访问map的元素。迭代器的基本思想是：迭代器变量中存储了数据结构的某个元素的位置，使得你能够访问该位置上的元素。通过调用迭代器提供的方法，可以继续访问数据结构中的下一个元素。

声明一个整型vector的迭代器需要用到特殊的语法，示例如下：

```
vector<int>::iterator
```

这个语法大意是说：现在有了一个整型的vector（vector<int>），我们还希望拥有一个能处理它的迭代器，因此用::iterator来表示。那么，迭代器要如何使用呢？由于迭代器中存储着数据结构的某个元素的位置，可以像这样来请求该数据结构的一个迭代器：


```
vector<int> vec;
vec.push_back( 1 );
vec.push_back( 2 );

vector<int>::iterator itr = vec.begin();
```

调用begin方法将返回一个迭代器，通过它能访问到vector的第一个元素。实际上，可以把迭代器看做一个指针——你可以通过它得到数据结构的某个元素的位置，亦可以使用它来访问该元素。回到刚才的例子，我们可以使用如下语法来访问vector的第一个元素：

```
cout << *itr; // 输出vector的第一个元素
```

这里对*运算符的使用，仿佛是在使用指针似的。这真是太棒了：迭代器跟指针一样，都是位置存储的一种方式。

要获得vector的下一个元素，只需要增加你的迭代器即可：

```
itr++;
```

这相当于命令迭代器前往vector的下一个元素。

也可以使用前缀运算符：

```
++itr;
```

这种做法在某些迭代器中效率会更高一些①。

通过对比当前的迭代器和末端迭代器，我们可以检查是否已经到达迭代遍历的结尾。调用迭代器的end方法可以获得末端迭代器：

```
vec.end();
```

因此，循环遍历整个vector的代码可以这样写：

```
for ( vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr )
{
    cout << *itr << endl;
}
```

① 其原因是，前缀运算符（++itr）先做增量，然后返回表达式的值，而如果你使用后缀运算符（itr++），它返回的是增量前的itr值，这意味着可能有保存旧值的需要。前缀运算符已经具有需要返回的值，因为它包含着运算的结果。

这段代码表示：创建一个迭代器，并获得整型vector的第一个元素；当前迭代器不等于末端迭代器时，继续对vector的迭代。输出每个元素。

我们要对这个循环做几个小小的改进。应该避免每次循环时都调用一次vec.end()：

```
vector<int>::iterator end = vec.end();
for ( vector<int>::iterator itr = vec.begin(); itr != end; ++itr )
{
    cout << *itr << endl;
}
```

实际上，可以将多个变量放到循环的第一个部分中，使代码看起来更整洁些：

```
for ( vector<int>::iterator itr = vec.begin(), end = vec.end(); itr != end;
++itr )
{
    cout << *itr << endl;
}
```

我们可以用非常相似的方法来遍历一个map。不过，map的一个元素里不仅仅只有一个值，而是两个：键和值。这样的话，该怎样使用map的迭代器呢？当你间接引用map的迭代器时，它有两个域：first和second。域first为键，而second为对应的值。

```
int key = itr->first; // 从迭代器中获得键
int value = itr->second; // 从迭代器中获得值
```

来看一段代码，它将map中的内容以较强的可读性输出出来：

```
void displayMap (map<string, string> map_to_print)
{
    for ( map<string, string>::iterator itr = map_to_print.begin(), end =
map_to_print.end();
        itr != end; ++itr )
    {
        cout << itr->first << " --> " << itr->second << endl;
    }
}
```

这段代码与遍历vector的代码极其相似，真正唯一的区别是map数据结构的使用和迭代器的first和second域的使用。

检查一个值是否在map中

有时候，你会想要检查给定的键是否已经存储在一个map中了。例如，如果你正在通讯簿中查找某人，可能想知道那个人是否真的在通讯簿中。这时，find方法正是你需要的。find方法返回一个迭代器：如果给定的键存在，则返回的是一个持有该键对应的对象位置的迭代器；如果给定的键不存在，返回末端迭代器。

```
map<string, string>::iterator itr = name_to_email.find( "Alex Allain" );
if ( itr != name_to_email.end() )
{
    cout << "How nice to see Alex again. His email is: " << itr->second;
}
```


另外，如果你尝试使用普通的方括号运算符访问一个map中不存在的元素：

```
name_to_email[ "John Doe" ];
```

那么，map会为你插入这个新的元素，对应的值为空。所以，如果你真的需要知道一个值是否在map中，请使用`find`方法；除此之外，可以安全地使用方括号运算符。

18.4 盘点 STL

我们还有很多STL的知识没有讲，但你现在已经掌握了充分利用STL类型的许多基础知识。`vector`类型是数组的完美替代品。当不需要太在乎插入和修改列表的时间开销时，vector也可以用来取代链表。只有在极少数高级应用，如文件输入输出的情况下，你会选择使用数组而不是vector。

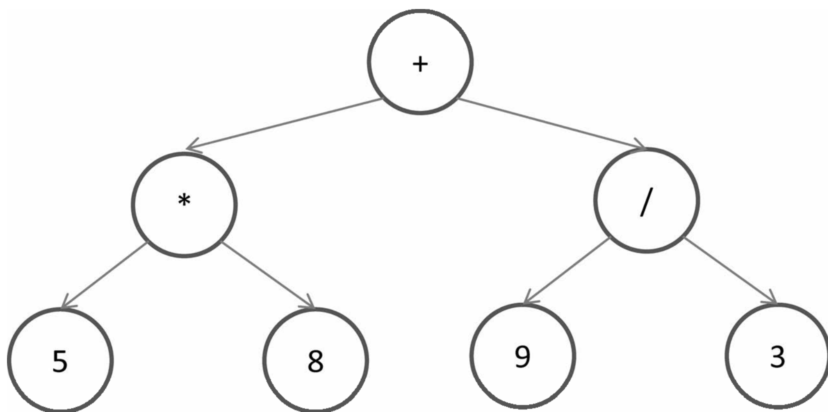
`map`可能是目前为止最好的一个数据类型了。我经常使用类似map的结构，它使得编写复杂的程序变得更自然，因为你不再需要担心如何创建许多的数据类型。相反，你可以专注于如何解决要解决的问题。在许多方面，map可以取代基本的二叉树——大多数情况下你不用实现自己的二叉树，除非为了特定的性能要求，或者真的需要使用树形结构。这就是STL真正厉害之处——大约80%的情况下，STL提供了核心的数据结构，因此你可以马上动手编写代码，解决特定问题；另外的20%，就是你需要知道如何自己构建数据结构的原因[①]。

有些程序员可能患有“非我发明”综合征——他们倾向于使用自己写的代码，而不是别人写的。在大多数情况下，你*不应该*自己去实现数据结构——自带的数据结构通常比自己写的要好，速度更快且更完整。但知道如何建立它们会让你更深入地了解如何使用它们，以及如何在确实需要的时候创建自己的数据结构。

那么，何时需要自己实现数据结构呢？假设你想写一个小型计算器，它可以让用户输入算术表达式并依照正确的计算顺序求出表达式的值。例如，读入5 * 8 + 9 / 3这样的表达式，然后计算它的值，计算顺序是先乘除、后加减。

事实证明，人们往往自然而然地想到采用树状结构来解决这个问题：

① 这个数据没有经过科学统计，而是我编造的。你的比例可能会与我的有出入，但我相信无论是哪个方向都不可能是100%。



以下面这两种方式来计算每个节点：

(1) 如果是一个数字，则返回它的值；
(2) 如果是一个运算符，则计算两个子树的值，并执行该运算。

建立这样一棵树，需要使用原始的数据结构。这时候，仅使用map是不够的。如果你唯一的工具是STL，就很难解决这个问题了。但如果你懂二叉树和递归，问题就简单多了。

18.5　进一步学习 STL

如果你想更多地了解STL，以下是一些很好的资源。

SGI是一个包含大量STL文档的网站：http://www.sgi.com/tech/stl/；斯科特・梅耶（Scott Meyer）的著作*Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*同样很精彩，它介绍了很多STL的概念和惯用语。网站http://en. cppreference.com/w/cpp也有很多关于STL元素的优秀文档。但它提供的不是STL的入门资料，而是C++标准库的实用参考材料。

18.6　问答题

(1) 什么时候适合使用vector？

A. 当你需要存储一个键和一个值之间的关联时
B. 当你为了最大限度地提高性能而需要改变元素的集合时
C. 当你不想关心数据结构进行更新的细节时
D. 就好像面试要穿西装一样，使用vector总是没错的

(2) 怎样从一个map中一次性删除所有元素？

A. 将元素设置为空　　B. 调用erase方法

C. 调用empty方法　　D. 调用clear方法

(3) 什么时候你应该实现自己的数据结构？

A. 当你速度要求很快的时候

B. 当你需要鲁棒性强的时候

C. 当你想要利用原始数据结构的优势时，比如建立一棵表达式树

D. 你永远不需要实现自己的数据结构，除非你喜欢这么干

(4) 以下哪一项正确地声明了一个vector<int>的迭代器？

A. iterator<int> itr;　　B. vector::iterator itr;

C. vector<int>::iterator itr;　　D. vector<int>::iterator<int> itr;

(5) 以下哪一项正在访问一个map的迭代器的键？

A. itr.first　　B. itr->first　　C. itr->key　　D. itr.key

(6) 怎样知道一个迭代器是否可用？

A. 跟NULL进行比较

B. 跟迭代的对象调用end()的结果进行比较

C. 检查它是否等于0

D. 跟迭代的对象调用begin()的结果进行比较

18.7　实践题

(1) 实现一个小型通讯录程序，用户可以输入姓名和电子邮件地址，删除或更新条目，以及显示通讯录中的所有条目。不用考虑将通讯录存到磁盘，程序退出时可以丢失数据①。

(2) 用vector来实现一个电子游戏的高分榜。分数可以自动更新，新的分值添加到榜中的正确位置。可以从上面提到的SGI网站中找到更多的对vector的操作。

(3) 写一个程序，它有两个选项：用户注册和用户登录。用户注册允许新用户创建一个登录名和密码。用户登录允许用户登录并显示两个选项：修改密码和退出。修改密码允许用户修改其密码，退出将使用户返回到原来的界面。

① 这只有在你既是程序员又是用户的情况下被允许。

第 19 章

更多关于字符串的内容

哇，我们已经学完了许多难懂的知识，恭喜你历经九九八十一难到达了这里！本章我们来休息片刻，不再学习新的数据结构了，而是回头看一个你已经知道的数据结构：字符串。尽管很不起眼，但字符串被随处使用，许多程序所做的工作几乎全是在读入和修改字符串。你经常要读入字符串然后显示给用户，并且也经常需要知道一个字符串的内容。例如，你可能想实现字符串的搜索功能，能够在字符串中查找某个特定的值。你可能想读入一串以逗号分隔的表格数据，实现一个高分榜，或者创建一个基于文本界面的冒险游戏。你每天最常用的一个应用——浏览器，很大程度上就是一个超大的字符串处理器，它处理各种HTML网页。所有这些问题都要求你除了能够读入和输出整个字符串外，还得会做其他工作。

字符串可能很大，占用极大的内存空间，因此我们可以利用之前学过的一些特性，来写出即使在函数之间传递字符串也能有很高效率的程序。具体来说，就是使用引用。本章将介绍各种可用于处理字符串的操作，以及讨论如何在使用这些操作时保持程序的高效率。在实践题中，你将有机会写一些有趣的字符串处理代码，学习到操纵字符串的强大能力。

19.1 读入字符串

当读取字符串到程序中时，有时候会想要读入一整行，而不是像以前一样，使用空格来做分隔符，这使得你每次只能读入一个单词。

一个特殊的函数getline，可以一次读取一整行。它接受一个“输入流”（input stream），从该输入流中读取一行文本。cin就是输入流的一个例子，你以前常常用它一次读取一个单词。（告诉你一个小秘密：cin其实是个对象，就好像string或者vector一样。它是一种称为输入流的类型，而cin>>是读入数据的方法。在第一章就把这一切都统统交代出来，似乎并不好！）

下面的程序将演示怎样从用户的输入中读入整行文本：

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main ()
{
    string input;
    cout << "Please enter a line of text: ";
    getline( cin, input, '\n' );
    cout << "You typed in the line " << '\n' << input;
}
```
示例代码41：getline.cpp

这个程序读入一行字符序列到字符串input中，直到遇到换行符（\n）——换句话说，直到用户按下回车键。

换行符本身将被丢弃，而只保存换行符之前用户的输入部分。如果你想保留字符串中的换行符，必须自己手动添加。你不仅可以使用换行符，还可以用任何需要的字符作为停止读入字符串的标记（这个字符称为“分隔符”，因为它标识了字符串读取的界限）。用户仍然需要按回车键让getline函数返回，但只有分隔符之前的文本被读入。

来看一个例子吧，这个例子演示了如何读取以逗号分隔的格式化文本（CSV格式）。CSV格式的数据看起来像这样：

```
Sam, Jones, 40 Asparagus Ave, New York, New York, USA
```

每个逗号分隔数据的一节，看起来就像一个电子表格，但和电子表格不同，它使用逗号来分隔列。让我们来写一个程序，读入用户输入的CSV数据，这些数据是网络游戏中玩家的名单，以这样的格式表示：

```
<player first name>,<player last name>,<player class>
```

学习了第28章后，你就可以对这个程序做一些修改，使得它能够从磁盘中读取CSV文件，但现在仅是读取用户输入的数据。first_name为空时，程序结束退出。

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    while ( 1 )
    {
        string first_name;
        getline( cin, first_name, ',' );

        if ( first_name.size() == 0 )
        {
            break;
```

```
        }
        string last_name;
        getline( cin, last_name, ',' );

        string player_class;
        getline( cin, player_class, '\n' );
        cout << first_name << " " << last_name << " is a " << player_class << endl;
    }
}
```

示例代码42：csv.cpp

注意，这里使用字符串的size方法，来检测一个字符串是否为空。这只是字符串中可用的众多方法之一。

19.2　字符串长度和访问单个元素

要查找字符串的长度，可以用length方法或你刚才见过的size方法。这两个方法都是string类的一部分，都能够返回字符串中的字符数：

```
string my_string1 = "ten chars.";
int len = my_string1.length(); // 或.size();
```

这里的size和length没有任何区别，任意选择一个你感觉自然的来用就好[①]。

字符串可以像数组一样被索引化。例如，你可以通过索引访问每个字符，故而遍历到字符串中的所有字符，就好像字符串就是一个数组一样。如果你想处理字符串中的单个字符，比如查找像逗号这样的特殊字符，此方法非常有用。

这时候，配合使用length或size方法很重要，这样你就不会试图越界访问字符串结尾后的内容。跟数组一样，越界访问字符串结尾后的内容是很危险的。

这里有一个小例子，演示如何循环一个字符串，并将它显示出来：

```
for( int i = 0; i < my_string.length(); i++ )
{
    cout << my_string[ i ];
}
```

19.3　字符串搜索与子字符串

string类支持简单的子串搜索和取子串操作，所用的方法是find、rfind以及substr。

① 这两种方法都存在的原因是：所有的STL容器对象都使用size方法，因此使用size可以保持一致性；但对大多数程序员来说，使用length来处理字符串更自然些。

find方法接受一个子串和原始字符串的一个位置，找到给定的子串从指定位置开始的第一个匹配项。其结果要么是返回该子串的第一个匹配项的索引，要么是一个特殊的整数值string::npos，表示没有找到该子串。

以下示例代码在给定字符串中搜索子串"cat"的每一个匹配项，并对匹配项的数目进行计数：


```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string input;
    int i = 0;
    int cat_appearances = 0;

    cout << "Please enter a line of text: ";
    getline( cin, input, '\n' );

    for ( i = input.find( "cat", 0 ); i != string::npos; i = input.find( "cat", i ) )
{
        cat_appearances++;
        i++;  // 向后移动一位,
              // 避免查找到同一个字符串
    }

    cout << "The word cat appears " << cat_appearances << " in the string "
<< '"' << input << '"';
}
```

示例代码43：search.cpp

如果你想从字符串的结尾开始查找子串，可以使用rfind方法，其使用方式与find几乎一模一样，所不同的是，rfind从给定的开始点向前搜索，而不是向后（字符串匹配仍然是从左到右，调用rfind搜索"cat"不会匹配到字符串"tac"）。

substr方法会创建一个新的字符串，它是原字符串从给定位置开始的给定长度的切片：

```
// 示例原型
string substr (int position, int length);
```

例如，要提取一个字符串的前10个字符，你可以这样写：

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
```

```
    string my_string = "abcdefghijklmnop";
    string first_ten_of_alphabet = my_string.substr( 0, 10 );
    cout << "The first ten letters of the alphabet are "
         << first_ten_of_alphabet;
}
```

19.4 通过引用传递

字符串可能很大，包含大量数据。当然，并不是每个字符串都会很大，但总体上，通过引用来接受字符串参数是个很好的习惯：

```
void printString (string& str);
```

回顾一下：引用参数跟指针类似，它不用复制原来的字符串变量，而是将该字符串变量的引用传递给函数：

```
string str_to_show = "there is one x in this string";
printString( str_to_show );
```

这里，printString函数并没有复制变量str_to_show，而是获得了该变量的地址；我们可以像使用原始字符串一样地使用参数str。

但引用传递可能有一个缺点：引用使得函数获得了原始变量的地址，所以在函数中可以修改该变量。虽然你第一次写这个函数时，可能不想改变传递进来的引用变量，但是，如果事后又回头去更新它，比如添加新功能之类的，你可能已经忘了，于是修改了引用变量的值，结果就会很糟糕。有人调用这个函数时会惊讶地发现：它们的数据被修改了！

C++提供了一种防止引用参数被意外修改的机制：在函数中可以指定一个引用为常量。const是C++中的特殊关键字，用来指定一个引用为常量。我们不可以修改const指定的引用参数，但可以读取。

```
void print_string (const string str)
{
    cout << str; // 合法，没有修改str
    str="abc" ; // 不合法！
}
```

每当你添加一个引用参数到函数中时，考虑清楚函数是否应该能够修改引用参数。如果你不希望修改这个参数，请将它标记为const，以确保函数不会且不能修改它。使用const可以很清楚地表明参数不会被修改。

const并不仅限于引用，你也可以用const来标记一个指针指向的内存。在这种情况下，程序可以写成类似这样的代码：

```
void print_ptr (const int* p_val)
```

```
{
    if ( p_val == NULL ) // 没问题，p_val指向的内存没有被修改
    {
          return;
    }
    cout << *p_val; // 没问题，内存访问没有修改内存
    *p_val = 20; // 出错，p_val指向的内存被修改了
    p_val = NULL; // 没问题，只是修改了指针本身，并没修改指针所指向的内存
}
```


注意，编译器是非常聪明的，它能分辨出代码是否在对一个指针所指向的内存进行赋值。它不仅仅只关注指针本身，还关注指针的间接引用正在做的事情。由于在参数传递时指针的值是被复制进来的，因此修改指针本身是完全合法的；改变p_val的值不会影响到传递到函数的原始变量。

const的使用还可以更广泛。可以用const来标记和强制任何给定的变量不会被修改。如果你试图修改它，编译器会告诉你，你正在做一件不打算做的事情。当你声明一个const变量时，必须立即给它赋值（因为再也不能够修改它了）。

```
const int x = 4; // 没问题，变量创建的同时进行赋值
x = 4; // 出错，x不能被修改
```

尽可能使用const是很好的编程风格。将变量标记为const使得剩下的代码更易于阅读，因为你知道没有人会修改它，所以一旦看到一个对该变量的赋值，就可以肯定它不会被改变。你不必跟踪这个变量有没有被赋予其他的值，从而专注于非const的变量正在发生的事情，它们是否被修改，等等。const还能确保你以后不会意外地修改这个变量，防止发生诡异的代码行为，原代码假定了该变量的值不会变，它应该跟开始时的值始终相同。

例如有一段代码，它提示用户输入名字和姓氏，然后创建一个包含用户的全名的字符串，你应该将全名变量标记为const，因为它不应该被改变。

19.4.1 const传播

const可能会像病毒一样传播。一旦将一个变量声明为const，你就不能通过引用将它传递给接受非const引用的函数，也不能通过指针传递给接受非const指针的函数，因为在函数中可能会试图通过指针修改该变量的值。const X*和X*是不同的类型，const X&（声明一个到X的引用）和X&也是不同的类型。我们可以将一个X*转换成constX*，或者将一个X&转换成const X&，反之却不行。例如，如果写了如下的函数，将会编译出错：

```
void print_nonconst_val (int& p_val)
{
    cout << p_val;
}
```

```
const int x = 10;

print_nonconst_val( x ); // 无法编译
// 不能将一个const int类型的变量传递给一个接受非const引用的函数
```

这一限制只适用于引用和指针，因为在引用或指针中，原始的值在函数中被共享了。如果变量是以复制方式传到函数中的，比如值传递，就不需要将函数参数标记为const了：

```
void print_nonconst_val (int val)
{
    cout << val;
}

const int x = 10;

print_nonconst_val( x ); // 没问题，x被复制一份到函数中
// val是print_nonconst_val函数的局部变量，因此，它是不是const并不要紧
```

因此，只要你把一个变量标记成const，就需要考虑下其他变量是否也需要标记成const，特别是函数的指针和引用参数。

使用const需要小心。如果你正在使用的库或辅助方法里没有使用const，可能会有些麻烦。从另一个角度来说，你应该在自己写的库或辅助方法里使用const，这样别人使用你的代码才不会因为const出问题。

C++的标准库考虑了const的问题，因此你可以放心地在自己的代码中将变量标记为const，并使用这些变量和标准库。

本书其余部分，我会适时地使用const变量。

还有一点要注意的是，你可以在循环内部声明一个变量为const，即使每次循环都修改（重置）该变量：

```
for ( int i = 0; i < 10; i++ )
{
    const i_squared = i * i;
    cout << i_squared;
}
```

变量i_squared可以声明为const，即使它每次循环都被修改。这是因为，变量i_squared的作用域范围在整个循环体里。从编译器角度来看，每次循环时变量i_squared都被重新创建。

19.4.2　const和STL

在关于STL的最后一章，我们展示了一个显示map的函数。你可能注意到了，map是以值传递方式传给函数的，这意味着整个map需要被复制一份传递到displayMap函数中。再来看一下

这个函数：

```
void displayMap (map<string, string> map_to_print) // map被复制了！
{
    for ( map<string, string>::iterator itr = map_to_print.begin(), end =
map_to_print.end();
          itr != end;
          ++itr )
    {
          cout << itr->first << " --> " << itr->second << endl;
    }
}
```

这里如果用引用的话，就更完美了。那样的话，我们就可以不用复制整个map，而是使用map的引用了。甚至，我们可以使用一个const引用，很清楚地表明这是一个纯粹的显示函数，不能以任何方式修改这个map。

```
void displayMap (const map<string, string>& map_to_print)
{
    for ( map<string, string>::iterator itr = map_to_print.begin(), end =
map_to_print.end();
          itr != end;
          ++itr )
    {
          cout << itr->first << " --> " << itr->second << endl;
    }
}
```

如果这样做的话，恭喜你，编译错误！出问题的原因是：通过将map标记为const，表明没有任何人能够修改map中的元素，但是，迭代器本身是允许修改map的。例如，下面的代码：

```
if ( itr->first == "Alex Allain" )
{
    itr->second = "webmaster2@cprogramming.com"
}
```

这段代码就通过迭代器修改了你的地址簿里我的地址。幸运的是，STL对const是很友好的，所有的STL容器都有第二种特别的迭代器类型，叫做const_iterator。你可以像使用普通的迭代器一样地使用const_iterator，除了不能够通过const_iterator来修改正在迭代的容器：

```
void displayMap (const map<string, string> map_to_print)
{
    for ( map<string, string>::const_iterator itr = map_to_print.begin(),
end = map_to_print.end();
          itr != end;
          ++itr )
    {
          cout << itr->first << " --> " << itr->second << endl;
    }
}
```

当被迭代的容器被标记为const时，你必须始终使用const_iterator来对其迭代。无论何时使用迭代器，都只用它来访问数据，而不要通过它来修改被迭代对象的内容，这是一种很好的习惯。

19.5　问答题

(1) 下面的代码哪些是合法的？

A. `const int& x;`
B. `const int x = 3; int *p_int = & x;`
C. `const int x = 12; const int *p_int = & x;`
D. `int x = 3; const int y = x; int& z = y;`

(2) 下面的函数签名中，哪一项可以让代码`const int x = 3; fun( x );`编译通过：

A. `void fun (int x);`　　B. `void fun (int& x);`
C. `void fun (const int& x);`　　D. A和C

(3) 判断一个字符串搜索没有成功找到目标元素的最好方式是？

A. 比较结果位置和0　　B. 比较结果位置和-1
C. 比较结果位置和string::npos　　D. 检查结果位置是否大于字符串长度

(4) 如何为一个const的STL容器创建迭代器？

A. 声明迭代器为const
B. 使用索引来循环遍历，不使用迭代器
C. 使用const_iterator
D. 声明模板类型为const

19.6　实践题

特别提醒：所有的实践题，都请尽量适时地使用const和const引用！也就是说，每当你写一个接受字符串的函数时，尽可能地通过const引用来传递该字符串。

(1) 写一个程序，它读入两个字符串，计数第一个字符串在第二个字符串中出现的次数。

(2) 写一个程序，允许用户输入类似CSV文件的表格数据，但它不是用逗号来做分隔符，而是由程序来检测有效的分隔符。首先，用户输入几行表格数据；接着，程序通过遍历输入数据中的所有非数字、非字母、非空格字符，检测出可能的分隔符。找到每一行出现的可能的分隔符后，显示给用户，并向用户询问使用哪一个作为分隔符。例如，如果用户输入：

Alex Allain, webmaster@cprogramming.com
John Smith, john@nowhere.com

应该提示用户从逗号“,”、@号和点号“.”中选择一个作为分隔符。

(3) 写一个程序，它读入用户输入的HTML文本。（别担心，我们以后会介绍如何从文件中读入文本。）它应该支持如下的HTML标签：<html>、<head>、<body>、、<i>以及<a>。每个HTML标签都有一个开始标签，如<html>，以及一个闭合标签，它有一个斜杠在前面，如</html>。标签里面是该标签控制的文本，比如“这里的文本是粗体的”，或者“<i>这里的文本是斜体的</i>”。<head></head>标签里面的文本是元数据，<body></body>里面的是要显示的文本。<a>标签用来表示超链接，里面有一个URL地址，按以下格式来表示：文本。

程序读入HTML文本后，应该简单地忽略掉<html>。然后，移除<head>部分的所有文本，它不应该在你的输出中出现。接着，程序应该显示出<body>里的所有文本，将和之间的文本以星号（*）围绕的方式显示，<i>和</i>里面的文本以下划线（_）围绕的方式显示，link text这样的标签内的文本以链接文本（linkurl）的形式显示。

第 20 章

使用Code::Blocks进行调试

现在，你已经学会了很多强大的编程技术，但是在更加复杂的程序中追踪bug[1]可能还是一件很困难的事情。幸运的是，一个称为调试器的工具可以帮你解决这些问题。调试器是一个用来检查程序运行状态的工具，它使得程序正在做的事情变得更容易让人理解。程序员新手往往排斥学习使用调试器，因为它看起来既烦琐又没必要。确实，为了使用工具而必须学习它是令人生厌的。但是不学习使用调试器是一种“捡了芝麻，丢了西瓜”的行为。调试器能节省大量时间，使用调试器就像放弃爬行，学习走路。你需要做一些练习，刚开始时会跌跌撞撞。但是当你适应之后，肯定会激动得“手舞足蹈”。

本章将介绍Code::Blocks的调试器。如果你使用的是Windows系统，应该已经安装Code::Blocks并且使用一段时间了。虽然有很多不同的调试器，然而这些概念都是相通的。我提供了大量的截图，这样即使你用的不是Windows系统，也能够理解本文的内容，看到编译器是什么样子的。几乎所有的开发环境都有自己的调试器，你的也不例外[2]。

贯穿本章，我将会使用有bug的程序来展示真实的调试过程。如果你想跟随每个调试过程的话，每个示例你都可以在Code::Blocks中创建该程序的新项目（或者在选择的开发环境中创建）。

下面是第一个程序，用来计算特定数额资金的利率（interest rate）、年利息（compounded annually）。遗憾的是，里面有一个bug导致程序输出有误。

```
#include <iostream>
using namespace std;

double computeInterest (double base_val, double rate, int years)
{
```

① bug，原意是“臭虫”或“虫子”，用来指代电脑系统或程序中，一些未被发现的缺陷或问题。——译者注

② 如果你用的是Linux系统，可以使用GDB；如果使用的是Visual Studio或Visual Studio Express，它们也都自带非常不错的调试器。还有许多其他的独立调试器可以使用，不过这些超出了本书的范围，例如WinDbg，它是微软的Windows系统调试工具的一部分：http://www.microsoft.com/whdc/devtools/debugging/default.mspx。苹果的Xcode也提供了一个调试器。

```
    double final_multiplier;
    for ( int i = 0; i < years; i++ )
    {
        final_multiplier *= (1 + rate);
    }
    return base_val * final_multiplier;
}

int main ()
{
    double base_val;
    double rate;
    int years;
    cout << "Enter a base value: ";
    cin >> base_val;
    cout << "Enter an interest rate: ";
    cin >> rate;
    cout << "Enter the number of years to compound: ";
    cin >> years;
    cout << "After " << years << " you will have " << computeInterest( base_val, rate,
years ) << " money" << endl;
}
```

示例代码44：bug1.cpp

这是该程序的运行结果：

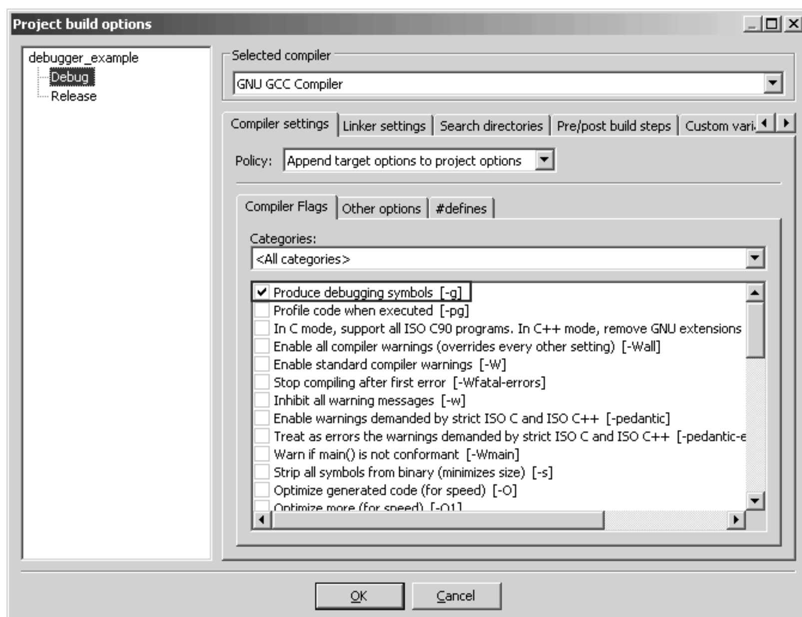
```
Enter a base value: 100
Enter an interest rate: .1
Enter the number of years to compound: 1
After 1 you will have 1.40619e-306 money
```

不对！`1.40618e-306`绝对是错误的资金金额！很明显，这个程序有bug。让我们尝试在调试器中运行程序，看看问题出自哪里。

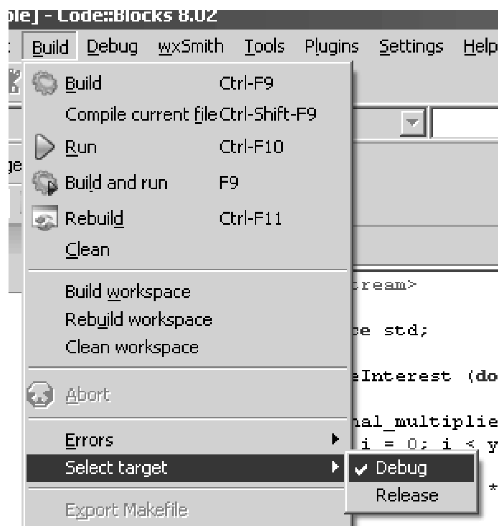
20.1 踏上调试之旅

首先，我们要确保Code::Blocks的配置正确，调试工作才能进行得更顺利。

为此，我们需要生成所谓的调试符号。调试符号可以让调试器知道代码的哪一行正在执行，这样你就可以知道程序运行到哪里了。为确保调试符号设置正确，请在Code::Blocks中选择项目|编译选项（Project | Build Options），会看到这样一个对话框：



你需要确保调试（Debug）目标里的生成调试符号（Produce debugging symbols）选项被勾选上。还需要在编译|选择目标|调试（Build|Select Target|Debug）中，确保调试（Debug）作为项目的目标被选中。



以上操作确保了目标是对项目进行调试，调试器将使用调试符号来编译你的程序。

如果你既没有调试（Debug）目标，也没有发布（Release）目标，也可以只勾选上生成调试

符号（Produce debugging symbols）选项作为当前的编译目标[1]，并且确保“从二进制中去除所有的符号（最小大小）[s]”（Strip all symbols from binary（minimizes size）[-s]）选项未选中。（通常，项目创建时默认生成这些编译目标类型。因此，确保你的配置正确的最简单方法是，保留项目创建时Code::Blocks的默认设置。）

万事俱备，只欠东风了。如果你的程序创建得比较早，但又必须改变它的配置，现在就先重建它吧。一切准备就绪后，我们可以开始调试了！

20.2 设置断点

调试器的价值在于，它能让我们看到程序正在做的事情——哪些代码正在执行，以及变量的值是多少。为了看到这些信息，我们需要“闯入”程序之中，不是那种入室抢劫的“闯入”，而是指我们让调试器暂停住程序的执行。为此，我们在程序的某个地方设置断点，然后在调试器下运行该程序。调试器将执行程序，直到到达设置了断点的代码行。此时，编译器便可以让你查看程序，或者一步步地执行程序，检查代码的每一行是如何影响你的变量的。

让我们在程序的前面，也就是main函数开始的地方，设置一个断点。这样，我们就可以查看整个程序的执行过程。要设置一个断点，先把光标移到这一行：

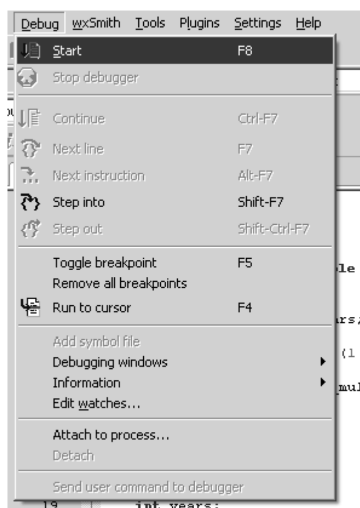
```
double base_val;
```

然后选择调试|设置断点（Debug|Toggle Breakpoint）或者按下F5。这会在该代码行旁边的侧边栏中设置了一个小红点，表明这一行有一个断点：

```
15      int main()
16      {
17 ●        double base_val;
18          double rate;
19          int years;
20          cout << "Enter a base value: ";
21          cin >> base_val;
22          cout << "Enter an interest rate: ";
23          cin >> rate;
24          cout << "Enter the number of years to compound: ";
25          cin >> years;
```

你可以使用设置断点命令或者单击小红点，来设置或取消设置该断点。现在，我们设置了一个断点，可以运行程序了！选择调试 | 开始（Debug | Start）或者按下F8。

[1] 如果使用的是g++，那么你需要在命令行参数中加上-g参数，以便让编译器能产生调试符号。如果你使用的是Xcode，它会自动生成调试符号。



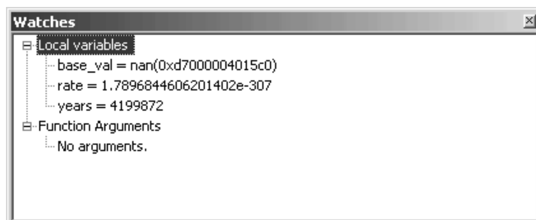
这样，程序将正常执行，直到遇到断点。在这个例子中，程序会立即达到断点，因为我们将断点设置在了程序的第一行。

现在应该看到了打开的调试器，它看起来大体是这样：

```
15      int main()
16      {
17 ●        double base_val;
18          double rate;
19          int years;
20 ▷        cout << "Enter a base value: ";
21          cin >> base_val;
22          cout << "Enter an interest rate: ";
23          cin >> rate;
24          cout << "Enter the number of years to compound: ";
25          cin >> years;
26
27          cout << "After " << years << " you will have " << computeInterest( base_val, rate, years ) << " money" << endl;
28      }
29
```

（可能也有其他窗口打开，我们稍后再讲。）首先要注意的是小圆点下面的三角形，它表示接下来要执行的代码行。它跟小红点之间相隔着若干行。它之所以没有紧挨着小红点，是因为变量的声明不产生任何的机器代码（机器代码是代码编译之后生成的可用于处理器执行的代码），因此，尽管断点看起来是在第17行，但实际上它在第20行（小圆点和三角形的左边的数字表示行号）。

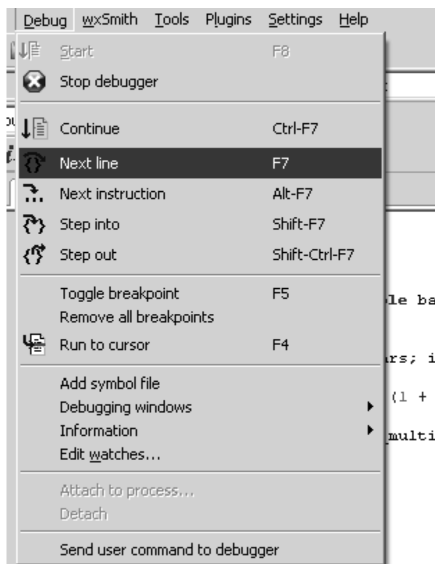
应该还有一个监视（Watches）窗口打开了，如下图（数字可能不同）：



如果你没有看到这个窗口，四处找找看。它可能躲在其他窗口后面。

我已经展开了监视窗口的两个子项：局部变量（Local variables）和函数参数（Function Arguments）。监视窗口会显示出所有当前可用的变量，包括局部变量和函数参数，以及这些变量的值。注意，这里的值看起来像乱码！这是因为我们还没有对它们进行初始化，这也是接下来的几行程序所要做的事情[1]。

为了执行接下来的几行代码，我们需要告诉调试器向下执行一行。所谓向下执行一行，就会执行当前的代码行（也就是三角形所标识的那一行）。Code::Blocks调试器调用下一行指令：



也可以按下F7，它是下一行指令的键盘快捷键[2]。

一旦走到下一行，程序就会执行`cout`语句，输出一条信息到屏幕中，要求你输入一个值。如果你尝试输入一个值，没任何效果——因为程序还在调试器的控制之下。让我们再次按下F7，执行下一行代码。按下F7后，程序会等待用户输入，因为这时候`cin`函数还未返回——`cin`函数需要在返回前得到用户的输入。继续输入值`100`（跟报告bug的输入用例相一致）。重复这一过程，分别提供输入值（同样，要跟报告bug的输入用例相一致）给接下来的两个变量，即输入`0.1`给利率，输入`1`给年数。

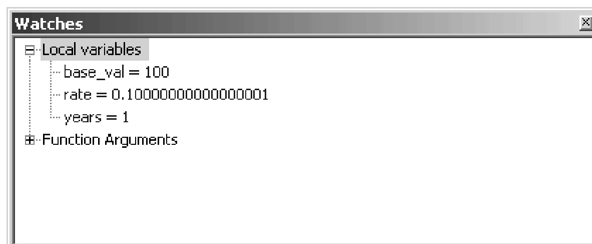
① 还记得变量声明时并不会初始化它们吗。

② 你可能会奇怪，为什么既有下一行（Next line），又有下一条指令（Next instruction）？我们将始终使用下一行。下一条指令用于没有调试符号的调试，超出本书讲述范围。

现在，到达了这一行代码：

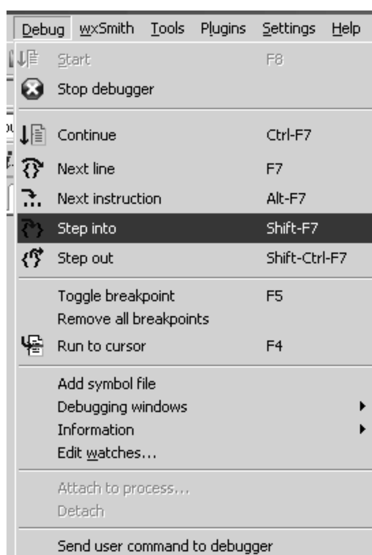
```
cout << "After " << years << " you will have " << computeInterest( base_val,
rate, years ) << " money" << endl;
```

再次确认输入是否正确。我们可以通过监视窗口来检查局部变量的值。



目前为止，一切都很好：base的值是100，rate的值是0.1，而years的值是1。你说什么？rate不是0.1？是的，它确实不是0.1，而是0.10000000000000001。不过，这最末尾的1只是浮点数的一种怪异的表示方式（还记得么，浮点数并不是精确的），它实在太小了，对大多数程序来说不会造成很大影响[①]。

现在，我们确定一切都没问题，来调查一下computeInterest函数中会发生什么。做到这一点的方法是使用另一个调试器命令，单步执行（Step into）：



单步执行会进入当前行的函数里面去执行，而不像下一行命令，只是执行函数然后显示给你

① 浮点错误可能会累积，在一些程序中引发严重问题。但在这个例子中，它的影响很小。

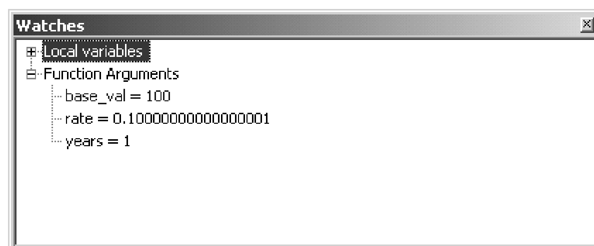
最终的结果，就如刚才在cin函数那里所见的那样。当你需要单步进入一个函数之中进行调试时，使用单步执行命令，就像我们接下来要做的一样。

单步进入computeInterest函数吧。可是，等等，你可能会奇怪：这条语句里有一堆的函数调用呀？

```
cout << "After " << years << " you will have " << computeInterest( base_val,
rate, years ) << " money" << endl;
```

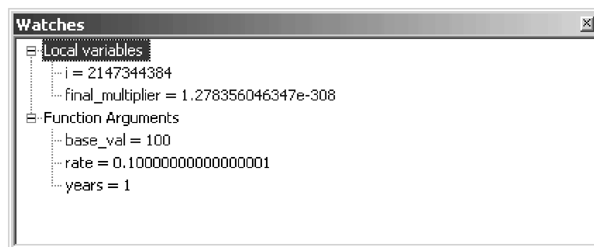

我们会单步进入cout函数吗？Code::Blocks调试器是很聪明的，它不会单步进入标准库的函数。在这个例子中，它会绕过那些我们毫无兴趣的函数（标准库的函数），直接单步进入computeInterest函数中。开始吧。

现在，我们进入了computeInterest函数之中。第一件事是确认函数参数是否正确——也许我们弄混了参数的顺序。请展开监视窗口里的函数参数部分：



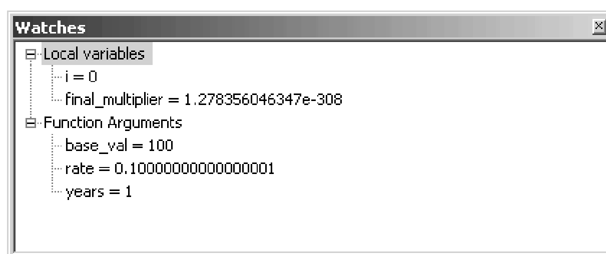
一切正常！

现在，来看看局部变量：



看到什么奇怪的东西了吗？变量i和final_multiplier的值根本就不对劲嘛！不过，别忘了我们上一次在监视窗口中也看到了局部变量的值很奇怪的情况，那是因为该变量还没被初始化。使用下一行命令（F7），执行循环语句，由于它与一些初始化操作相关联，我们看看会发生什么。

循环中的初始化操作只有一行，因此现在可以再次检查局部变量。此时，这两个变量看起来是这样的：



好的，i的值很好，不过，final_multiplier的值是怎么回事儿？它看起来没有正确初始化啊。而且，接下来要执行的语句即将要用到final_multiplier：

```
final_multiplier *= (1 + rate);
```

这条语句的意思是，将final_multiplier乘以(1 + rate)，再把结果重新赋值给final_multiplier。但是我们看到final_multiplier并没有被初始化，因此这个乘法的结果将会是个莫名其妙的值。

如何修复这个bug呢？

我们需要在声明final_multiplier变量的语句中，把它也初始化。在这个例子中，它应该被初始化为1。

就是这样，我们发现了问题并解决了它。太感谢你了，编译器！

20.2.1　调试崩溃问题

让我们来看看另一种bug——程序崩溃。崩溃往往是新手程序员最害怕的，因为它们看起来似乎很严重。但随着时间的推移，崩溃将会成为你最喜欢追踪的bug。这是因为，你会很容易找到发生问题的位置。程序崩溃是因为数据损坏，所以你可以在程序崩溃的地方停止下来，调查出到底是哪个数据出现了问题，然后找到问题根源。

下面是一个简单但有问题的程序，它创建了一个有两个节点的链表，然后输出列表里的每个值。

```
#include <iostream>

using namespace std;

struct LinkedList
{
    int val;
    LinkedList *next;
};

void printList (const LinkedList *lst)
```

```
{
    if ( lst != NULL )
    {
        cout << lst->val;
        cout << "\n";
        printList( lst->next );
    }
}

int main ()
{
    LinkedList *lst;
    lst = new LinkedList;
    lst->val = 10;
    lst->next = new LinkedList;
    lst->next->val = 11;
    printList( lst );

    return 0;
}
```

示例代码45：bug2.cpp

当你运行这个程序时，很遗憾，出问题了。它可能会崩溃，或者进入一个无限循环。总之，有些事情不对劲！

让我们在调试器里运行它，看看能否有所帮助。选择调试|开始（Debug|Start），或者按下F8。

几乎是与此同时，调试器弹出一条消息：



发生了一个段错误。段错误发生于不合法的指针使用。通常，这意味着程序试图引用一个空指针（NULL）或不合法的指针（要么是一个已经释放的指针，要么是一个从未初始化过的指针）。你可以把它想象成是程序试图访问一段它没有获得的内存[①]。

我们要怎样才能找出错误指针是从何而来的呢？想一想，编译器在崩溃发生的代码行中停止了。我们单击对话框中的OK，然后，找到程序中的三角形那一行，看看发生崩溃的代码行：

```
cout << lst->val;
```

这一行代码中只有一个指针，即lst。让我们使用监视窗口，看看lst的值是多少。从监视

① 在某些环境中会使用术语访问冲突（access violation），意思是一样的。

窗口中，我们可以看到，`lst`的值是`0xbaadf00d`[1]！很奇怪的数字，不是吗？这其实是编译器用来初始化内存分配时使用的一个特殊值。这一特性只有在编译器下运行时才会使用，这就是为什么你在编译器下运行程序跟在编译器外运行可能会看到不同的行为的原因。编译器使用一个一致的值来获知你访问了一个不合法的指针，从而立即发起一个段错误[2]，帮助你进行调试。

现在，我们知道了`lst`尚未初始化。但是，为什么它没有被初始化呢？来使用编译器的另一个功能——*调用栈*（call stack）。调用栈显示了当前所有正在执行中的函数。以下是在窗口中看到的调用栈的样子：

Call stack

Nr	Address	Function	File	Line
0	004013FD	printList(lst=0xbaadf00d)	c:/l/debugger...	15
1	00401431	printList(lst=0x3e24e8)	c:/l/debugger...	17
2	00401431	printList(lst=0x3e2438)	c:/l/debugger...	17
3	004014A0	main()	c:/l/debugger...	28

其中有几列：编号（Nr）是用来指代每个栈帧的数字；地址（Address）是函数的地址[3]。函数（Function）是函数的名称及其参数（事实上，你只通过调用栈也可以看到`lst=0xbadf00d`）；最后还有文件（File）和代码行（Line），以便让你找到正在执行的代码行。

调用栈顶部的函数即为当前正在执行的函数，下面的函数调用了它，以此类推。调用栈最底部的函数是main函数，因为它是程序开始执行的第一个函数。

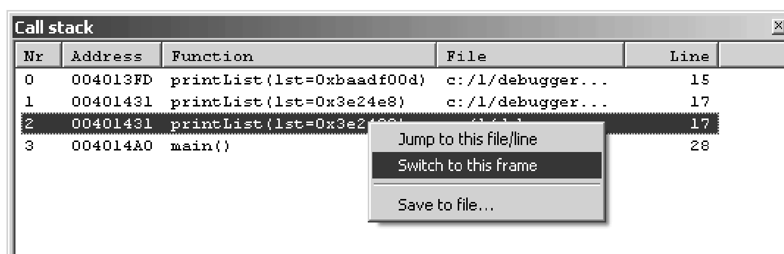
可以看到，对函数`printList`的调用有三次。前两次调用所拥有的指针是合法的，而第三次调用的指针值是`0xbaadf00d`。还记得么，`main`函数在列表中创建了两个节点。前两次调用`printList`一定是在使用这两个节点，而第三次调用使用的是一个未初始化的指针。我们现在再看看初始化列表的代码，发现从来没有为列表末尾的节点设置指向下一个节点的值为`NULL`。

虽然这个问题解决了，不过你有时候还会想要找到不同的栈帧的更多信息。你可以切换调试器的上下文到任何一个栈帧，以便检查其局部变量。请右键单击你感兴趣的栈帧，并选择切换到此栈帧（Switch to this frame）：

① 你可能对这个语法不熟悉，它是一个十六进制数字。十六进制数通常使用0x为前缀，并使用字母A ~ F来表示数字10 ~ 15。因此，十六进制的0xA跟十进制的10是一样的。

② 另外一种替代方案是，使用以前存储在变量中的值来定位指针的存储位置。由于内存是不可预测的，它甚至可能看起来就是合法的，这会使得程序行为非常奇怪，因而难以追踪。例如，程序不是立即崩溃，而是读到一些不该访问的内存，然后在后来使用该内存时才崩溃。编译器通过使程序行为一致和确保程序尽可能早地崩溃，这样你就能够尽可能地接近原始的问题，事情就变得简单多了。

③ 当你对程序进行汇编级别的调试时，函数地址能派上大用场。但大多数情况下，你不太需要用到它。



调试器将移动三角形，向你显示该栈帧正在进行函数调用的地方。这时你还可以使用监视窗口检查这个栈帧的局部变量。

20.2.2　强行进入一个“悬停”程序

有时候，你碰到的不是简单的崩溃，而是程序“被困住了”——可能是进入了一个无限循环，也可能是在等待一些耗时的系统调用完成。遇到这种情况，你可以让程序在调试器下执行，等遇到该问题时，让调试器“强行进入”该程序中。

使用一段示例代码来看看要怎么做：

```
#include <iostream>

using namespace std;

int main ()
{
    int factorial = 1;
    for ( int i = 0; i < 10; i++ )
    {
        factorial *= i;
    }
    int sum = 0;
    for ( int i = 0; i < 10; i++ )
    {
        sum += i;
    }
    // 剔除了2的阶乘
    int factorial_without_two = 1;
    for ( int i = 0; i < 10; i++ )
    {
        if ( i == 2 )
        {
            continue;
        }
        factorial_without_two *= i;
    }
    // 剔除了2的求和
    int sum_without_two = 0;
    for ( int i = 0; i < 10; i++ )
```

```
        {
            if ( i = 2 )
            {
                continue;
            }
            sum_without_two += i;
        }
    }
```

示例代码46：bug3.cpp

当你运行这个程序时，它永远不会退出。它在某些地方“被困住了”。为了找到这个地方，我们将让它在调试器下运行，等到它被卡住时，再检查。

首先，编译程序并将它在调试器下运行（选择Debug | Start或按下F8）。一旦程序开始运行，你会发现它不会退出；你应该在某些地方被困住了，大概是某种无限循环。让调试器强行进入正在执行的程序，这样我们就可以看到正在发生的事情。为此，我们选择调试 | 停止调试器（Debug | Stop Debugger）。停止调试器会导致调试器强行进入该程序，让你查看当前执行点的信息。（如果程序已经在调试器中运行的话，你也可以用这种方法来结束调试会话。）

一旦停止程序，你应该看到调用栈变成这个样子（不过，这个例子的调用栈看起来非常地奇怪）：

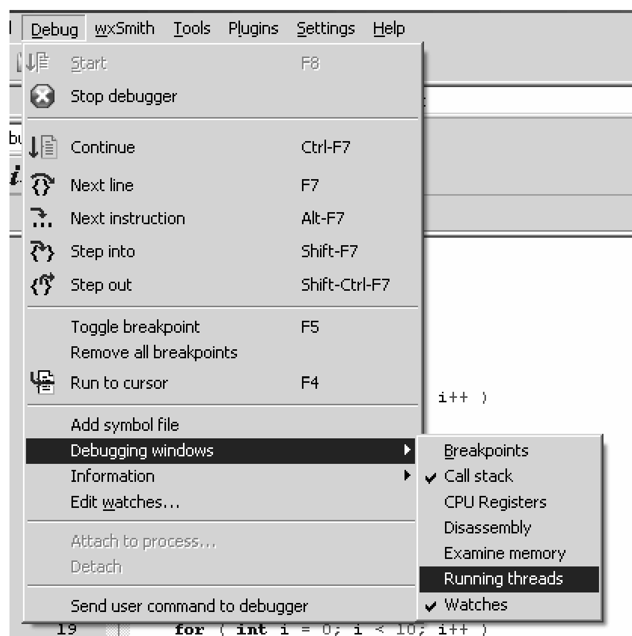
Call stack

Nr	Address	Function	File	Line
0	7C90120F	ntdll!DbgUiConnectToDbg()	C:\WINDOWS\sy...	
1	7C951E40	ntdll!KiIntSystemCall()	C:\WINDOWS\sy...	
2	00000005	??()		
3	00000004	??()		
4	00000001	??()		
5	004EFFD0	??()		
6	00000000	??()		

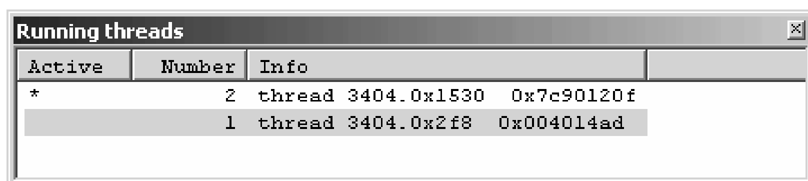
这里根本没有我们的代码！到底是怎么回事儿呢？你所看到的是“强行进入”一个正在执行的程序所导致的结果。注意到该调用栈的顶部是`ntdll!DbgUiConnectToDbg`了吗？`ntdll`是Windows核心的动态链接库，正在被调用的函数（即`DbgUiConnectToDbg`）被用来强行进入一个正在运行的进程。那么，我们正在执行的程序代码在哪里呢？事实是，为了强行进入一个进程，编译器创建了另外一个线程（线程是一种并发执行代码的方式）。为了闯入该进程，编译器需要能够在我们的原始代码执行的同时，执行一些其他代码。它通过创建一个新线程来执行闯入进程的代码，从而做到这一点。我们在前面的例子中没有碰到第二个线程，是因为该进程一开始就设置了断点，所以调试器有足够的控制力，不用创建第二个线程。在这个例子中，我们想要在某个时间点强行进入该程序，以便找出正在执行的代码，而不是在某一行特定代码设置断点。现在为了找到自己的代码，我们需要切换到正确的线程中。

为了切换线程，我们需要调出线程窗口。选择调试|调试窗口|运行的线程（Debug | Debugging

windows | Running threads ）：



在线程窗口中，我们看到两个线程：



在活动（Active）这一列中，使用*表示当前线程。在这个例子中，当前线程即为用来强行进入进程的线程。我们需要切换到其他线程，以便查看关于它的信息。为此，右键单击其他线程，选择切换到此线程（Switch to this thread）：



现在，可以回到调用栈，看到更多可以理解的信息：



这就是我们的代码了。你将看到，调试器将三角形放在了第29行，指示接下来要执行的代码行。以下是此段代码：

```
for ( int i = 0; i < 10; i++ )
{
    if ( i = 2 )
    {
        continue;
    }
    sum_without_two += i;
}
```

由于程序被困住了，而且现在在一个循环中，一个很自然的猜想是，这个循环可能没有终止。怎样证明这个猜想呢？我们一起来看看程序。

不过要小心一点。如果我们只是在调试器中进行下一行这个命令，它将会执行那个用来强行进入进程的线程的代码，因为它是当前的执行线程。不要执行“下一行”这个命令，而是在自己的代码中设置一个断点，然后让程序运行，直到它到达断点。[1] 我们在`if`语句这一行放置一个断点，然后继续执行程序（Ctrl-F7）。当程序遇到断点而停止时，我们已经在正确的线程中了，这时你可以使用下一行这个命令单步执行程序，看到程序正在发生的事情。

你将总是碰到`if ( i = 2 )`这条语句，然后回到循环的起始点。

到底怎么回事儿呢？来看看监视窗口里的局部变量`i`的值。在执行到循环体之中时，`i`的值为`2`。执行完循环代码后，`i`的值为`3`。接着，当我们执行`if`语句这一行时，你会发现，`i`的值又回到了`2`。

看来是有人总在把`i`的值设置成`2`——这里肯定是`if`语句无疑。事实上，这是一个常见的等号错误，这里应该用双等号。

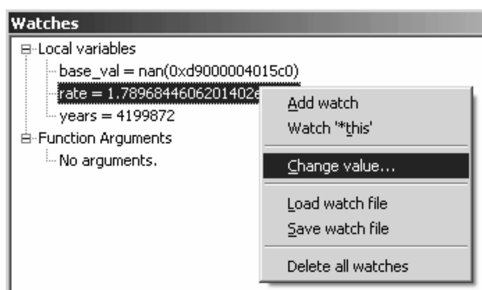
顺便提一句，你可能会奇怪：为什么程序从来没有真正到达过`continue`这一行，为什么它只是直接从`if`语句跳回到`for`循环？这是编译器搞的鬼——有时，一行特定的代码很难有直接匹配的机器代码。在这个例子中，编译器很难区分`if ( x = 2 )`语句和`continue`语句。你以后会时不时看到编译器中执行的代码仿佛跟期望的不一样。当你调试时，会开始注意到这些特殊的

① 有些调试器允许你通过“冻结”线程，选择控制哪个线程为正在运行。但Code::Blocks没有这个选项。

情况，比如刚才遇到的这个。

20.2.3 修改变量

调试时，有时候你可能希望修改变量的值——例如为了确认如果把一个变量设置为特定值后，剩下的代码就能够正常地工作。你可以使用监视窗口做到这一点：右键单击一个变量，选择改变值（Change Value），然后任意设置想要的值。



但要注意，不要在变量马上要被初始化或被覆盖之前做这件事。

20.2.4 总结

Code::Blocks是一个可以使你迅速上手的调试器。如果你使用的是非Windows的系统，许多相同的概念同样适用，虽然形式可能稍微不一样。调试的基本思路是：了解更多程序的状态，使用像断点这样的工具，逐句穿过程序到达合适的位置，然后通过了解调用栈和各个不同变量的值来理解程序正在做的事情。

20.3 实践题

与其他章不同，本章不测验与调试相关的问题，或让你写程序，而是提供一些有问题的程序让你调试。每个程序都存在一些不当行为。你应该在Code::Blocks 中给每个程序创建新的项目，并调试它们。某些程序不止有一个bug。

20.3.1 问题 1：指数问题

```
#include <iostream>

using namespace std;
int exponent (int base, int exp)
{
    int running_value;
```

```
    for ( int i = 0; i < exp; i++ )
    {
          running_value *= base;
    }
    return base;
}

int main()
{
    int base;
    int exp;

    cout << "Enter a base value: ";
    cin >> base;
    cout << "Enter an exponent: ";
    cin >> exp;
    exponent( exp, base );
}
```

示例代码47：practice1.cpp

20.3.2　问题 2：相加问题

```
#include <iostream>

using namespace std;

int sumValues (int *values, int n)
{
    int sum;
    for ( int i = 0; i <= n; i++ )
    {
          sum += values[ i ];
    }
    return sum;
}

int main()
{
    int size;
    cout << "Enter a size: ";
    cin >> size;
    int *values = new int[ size ];
    int i;
    while ( i < size )
    {
          cout << "Enter value to add: ";
          cin >> values[ ++i ];
    }
    cout << "Total sum is: " << sumValues( values, size );
}
```

示例代码48：practice2.cpp

20.3.3　问题 3：斐波那契程序的bug[①]

```
#include <iostream>

using namespace std;

int fibonacci (int n)
{
    if ( n == 0 )
    {
        return 1;
    }
    return fibonacci( n - 1 ) + fibonacci( n - 2 );
}

int main()
{
    int n;
    cout << "Enter the number to compute fibonacci for: " << endl;
    cin >> n;
    cout << fibonacci( n );
}
```

示例代码49：practice3.cpp

20.3.4　问题 4：列表的错误读取和错误输出

```
#include <iostream>

using namespace std;

struct Node
{
    int val;
    Node *p_next;
};

int main()
{
    int val;
    Node *p_head;
    while ( 1 )
    {
        cout << "Enter a value, 0 to replay: " <<endl;
        cin >> val;
        if ( val = 0 )
        {
            break;
        }
        Node *p_temp = new Node;
```

① 如果不熟悉斐波那契数列，请查阅：http://en.wikipedia.org/wiki/Fibonacci_number。

```
        p_temp = p_head;
        p_temp->val = val;
        p_head = p_temp;
    }
    Node *p_itr = p_head;
    while ( p_itr != NULL )
    {
        cout << p_itr->val << endl;
        p_itr = p_itr->p_next;
        delete p_itr;
    }
}
```

示例代码50：practice4.cpp

Part 3 第三部分

编写大规模程序

注意：如果你从本书的开头一直看到现在，而连一道实践习题都没有做，那么先停下来吧。如果你没有写过一些代码的话，根本无法理解并运用这一部分的知识。你将要接触的是本书中最重要的一些知识，但对于没有实践经验的人这些知识没有意义。

从开始到现在我们已经介绍过很多概念，你可以借助它们来实现新的想法。而现在，我们不仅要讨论如何实现新想法，还要介绍如何实现更大规模的程序。到目前为止，你还只是写过一些很短的程序，我猜大部分不超过几百行。当程序规模不是很大的时候，还好，你能凭大脑记住它们；但是，你可能已经发现越长的代码处理起来越困难。就算你还没有注意这点，也将会在某个时候遇到程序规模太大的情况。对有些人来说几百行就算长了，对有些人来说则是几千行或者更多，然而你记得再多也没用，好记性是个不错的技能，但是没人可以光凭记忆就能完成任何想要做的事。所有的程序都会随着规模的增大变得无法凭记忆完全理解。打算写个游戏？写个科学仿真程序？还是写个操作系统？你需要掌握一些更容易设计和理解大规模程序的技巧。

幸运的是，很多程序员涉足这一领域并且开发出了让我们更容易构建大规模程序的技术。接下来几章中介绍的原理就能让我们写出规模更大、复杂性更高的程序。这些理论同样可以让小程序的设计更加简单。

让我们先来研究一下几个在讨论如何设计大规模程序的时候会反复用到的概念。我们将从代码本身开始介绍；也就是如何在硬盘上存放代码，让它不只是单个庞大的 C++ 文件。然后，我们会讲述程序的逻辑设计：如何才能够在写程序的时候不再需要记住每段代码实现的功能细节。

第 21 章

将程序分解

21

当程序规模越来越庞大的时候，你不会愿意让整个程序都在单个源文件中。要做改动会很困难，而且当你要找某个东西的时候会找不着北。一旦程序达到数千行的时候，你肯定想要把它们分开放到几个不同的源文件中[①]。

使用多个源文件，你会更容易知道要找的东西在哪里，因为每个文件都相对较小，并且所包含的代码也是和程序的某个功能相关的。多个源文件同样便于程序的设计，因为每个头文件都只包含了与它相关的源代码的接口声明，这样别的程序就不能调用那些没有在它们头文件中声明的方法或者数据结构。这听起来可能像是个限制，但是在现实中，它可以让你更容易把程序的每个子系统自身和那些它们提供给别的子系统的功能分开。

21.1　理解 C++的构建过程

在将代码分解到不同的文件中去之前，你需要更多地了解一下C++的编译过程。

其实，编译这个说法并不准确——编译甚至都不包含生成一个可执行文件。生成一个可执行文件是需要好几个步骤的过程；最重要的步骤是*预处理*、*编译*和*链接*。从源代码到可执行文件的整个过程最好用*构建*这个词来表示。编译只是构建过程中的一部分，不能代表整个构建过程。但是，你会经常看到有人用编译这个词来表示整个过程。通常情况下，你不需要为每个步骤都单独执行一个命令——比如编译器就会自动调用预处理程序。

21.1.1　预处理

构建过程的第一步就是编译器运行C语言的*预处理程序*。C语言预处理程序的目的是在编译之前对源文件做一些文本的替换。预处理程序能够理解预处理指令，*预处理指令*（preprocessor directive）就是那些直接写到源文件中的命令，它们由预处理程序来处理而不是编译器。

① 我曾经有一次要处理一个有20 000代码大小为0.5 MB的源文件。没人愿意碰它！

所有的预处理指令都以磅的符号（#）开头。编译器从头到尾都不会看到预处理指令！

例如，下面的这句声明：

```
#include <iostream>
```

它告诉预处理程序直接把iostream中的内容放到当前的文件中。每次你包含了一个头文件，这个头文件都会在编译器看到它之前被全部复制到当前的文件中，并且#include指令会被移除。

预处理程序同样会展开宏指令。宏指令就是一串文字，它将会被别的内容代替，这些代替它的内容通常比宏指令更复杂一些，是一大串字符。宏指令可以让你将常量放在唯一的、有中心的地方，以便于更方便地修改。

例如，你可以这样写：

```
#define MY_NAME "Alex"
```

然后可以在整个源文件中用MY_NAME来代替"Alex"。

```
cout << "Hello " << MY_NAME << '\n';
```

编译器看到的就是：

```
cout << "Hello " << "Alex" << '\n';
```

如果你要修改名字，那么只需要修改包含#define的那一行代码，而不是必须对全部的代码来个查找/替换。宏指令把一些信息集中到一个地方，这样你在修改它们的时候就更方便了。如果想要给程序加个代码中任何地方都可以引用的版本号，可以用宏指令来定义它：

```
#define VERSION 4
// ...
cout << "The version is " << VERSION
```

因为预处理发生在代码编译之前，所以它还可以用来移除代码——有时你想要能够让某些特定的代码只在调试构建的时候才被编译。为此，你可以告诉预处理程序，只有在某个宏指令定义了的情况下才包含某段代码。然后，你想要保留这段代码，就定义个宏指令，如果不想要这段代码，就删掉相应的宏指令。

比如，你可能有些用来调试的代码会输出一些变量的值，但是不想让这些代码在任何时候都不停地输出。你可以实现这样的效果，让调试代码在一定条件下才被包含进构建的过程中。

```
#include <iosteam>

#define DEBUG

using namespace std;
```

```
int main ()
{
    int x;
    int y;
    cout << "Enter value for x: ";
    cin >> x;
    cout << "Enter value for y: ";
    cin >> y;
    x *= y;

#ifdef DEGUG
    cout << "Variable x: " << x << '\n' << "Variable y: " << y;
#endif
    // 接下来继续使用x和y
}
```

示例代码51：define.cpp

如果想要关闭变量值的输出，只要把`#define DEBUG`注释掉就可以了：

```
// #define DEBUG
```

C预处理程序还支持检测某个宏指令是不是没有定义过。例如，你可以用`#ifndef`（是否未定义）指令，来执行只有在DEBUG没有被设置的情况下才需要执行的代码。我们在讲到使用多个头文件的时候会用到这个知识。

21.1.2　编译

编译是指将源文件（a.cpp）转换为目标文件（a.o或者a.obj）。目标文件以一种计算机处理器能够理解的形式包含了你的程序，也就是机器语言指令，源代码中的每一个函数都在其中。每一个源文件都会单独编译，这表示对应的目标文件只包含编译过的源文件所对应的机器语言。举个例子，如果你编译了（不包括链接的步骤）三个独立的源文件，将得到三个输出的目标文件，每个的名字都是.o或者.obj（扩展名取决于所用的编译器）。这些文件每一个都含有从相应的源文件翻译来的机器语言。但是现在还无法运行它们。你需要把它们转换成操作系统能够使用的可执行文件。这时候链接器就派上用场了。

21.1.3　链接

链接是指使用一堆目标文件再加上库文件来生成一个可执行文件（比如一个exe或者DLL文件）。[①]链接器以相应的格式先创建一个可执行文件，然后把每个目标文件的内容转移到这个可执行文件中去。链接器还会处理那些引用了源文件中没有定义过的函数的目标文件。例如，目标文件中引用了C++标准库当中的函数。每当你调用C++标准库的时候（如`cout << "Hi"`），就是在使用自己代码中没有定义过的函数。库函数当中的函数同样是定义在目标文件中的，只不过这些

① 或者，如果你只有一个源文件的话，就只有一个目标文件。链接的步骤一直存在，哪怕是只有单个文件的简单程序。

目标文件不是你自己写的，它们是编译器厂商提供的。在编译的时候，正因为你引入了iostream头文件，所以编译器知道对库函数的调用是合法的，但是又由于这些函数不属于你的cpp文件，所以编译器在调用它们的地方仅仅留下一个标记。链接器会把所有的目标文件都过一遍，在每个编译器留下过标记的地方，它会找到正确的该调用函数的地址然后用这些链接过来的其他文件中的正确的地址把编译器留下的标记都替换掉。

这步操作有时称为整理。当把程序分散到不同的源文件中时，你就是利用了编译器可以整理所有调用其他源文件中的函数的功能。如果链接器找不到某个函数的定义，那么它就生成一个未定义函数的错误，即使代码通过了编译器那关，也不能表示它们没有错误。在链接器那里，整个程序会第一次被以一种能够发现上述问题的方式查看一遍。

21.1.4　把编译和链接分开的原因

因为不是每个函数都需要定义在同一个目标文件中，将所有的源文件一次编译，之后再把它们链接起来是可行的。如果你改变了其中的某一个文件（FrequentlyUpdate.cpp），但是没有修改另一个（InfrequentlyChanged.cpp），那么InfrequentlyChanged.cpp所对应的目标文件并不需要重新编译。在构建项目的时候跳过那些不必要的编译可是省下大量的时间。代码量越大，可以省下的时间就越多[①]。

要想获得按条件编译的最大好处，你需要一个可以记住某个特定的目标文件是否失效的工具，也就是在上次编译之后改变了这个目标文件所对应的源文件（或者改变了该源文件所包含的某个头文件）。如果是在Windows上并且使用Code::Blocks，那么你已经有了这一功能。如果用的Mac，那么XCode在你通过File|New|New file...新建文件的时候会自动处理这个问题。如果用的是Linux，你可以使用一个叫make（http://www.gnu.org/software/make/make.html）的工具，大部分版本的*nix系统都自带的[②]。

21.2　如何把程序分开到不同的文件中

那么你该如何组织代码来利用分开编译的优势呢？让我们来看看一个在程序Orig.cpp中有公共代码的简单例子，你现想要在一个新的程序中重用它。我将以一种按部就班的方式来描述这一过程，这样你可以看到每一步操作，然而在现实当中好多步骤是可以一次搞定的。

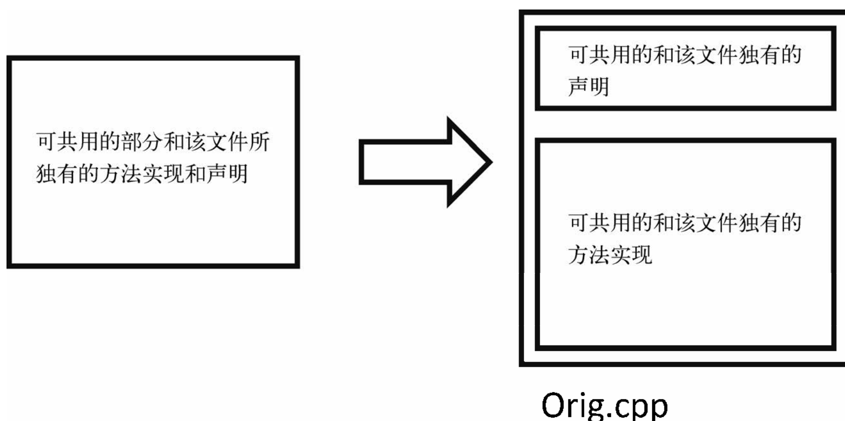
21.2.1　第一步：将声明和定义分开

如果没有试过把代码分到不同的文件中去，你可能在函数的声明和函数的定义之间没有一个

① 我亲眼见过需要好几个小时从头编译的源代码，并且听说过需要几天才能编译完成的源代码。

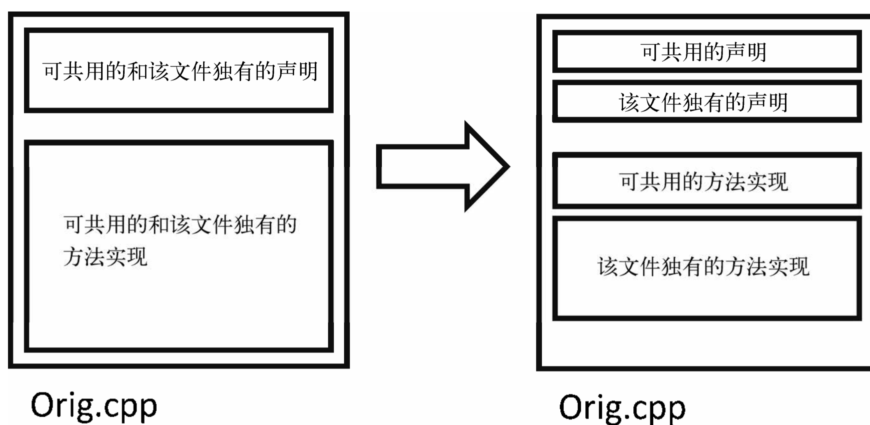
② 可以到这里获得更多的关于makefiles的知识：http://www.cprogramming.com/tutorial/makefiles.html。

明显的界限，那么第一步就是确保所有的函数都有对应的声明，然后把这些声明移到文件的最上面，看上去就像这样：



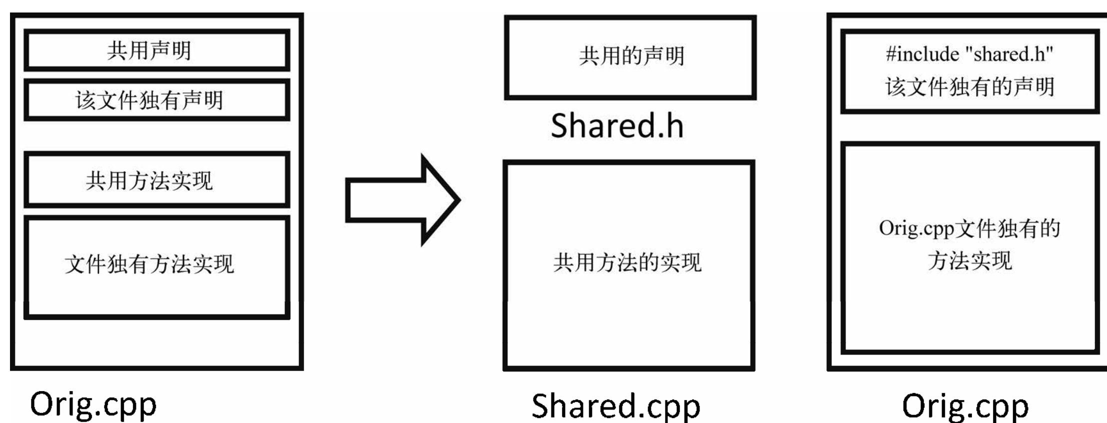
21.2.2 第二步：找出哪些函数需要共享出去

现在函数声明和函数定义已经分开了，你可以过一遍并找出哪些是这个文件独有的，哪些应该放在公共文件中。



21.2.3 第三步：把共用的函数移到新的文件中

现在你可以把共用的声明移到一个新的文件Shared.h中去了，共用的函数实现也可以移到Shared.cpp。同时，你需要在Orig.cpp中引入Shared.h。你可以继续调用那些共用的函数，因为它们的声明都在Shared.h中了。你需要像上述这样来配置，然后构建Orig.cpp时，程序就会把目标文件Shared.obj也链接进来。我们在下面描述一下这些操作的细节。



21.2.4 看一个完整的例子

下面是一段实现了通用链表的小程序，正好写在了Orig.cpp文件中。我们就选择这段代码，并且把它分成一个可以重用的头文件和源文件。

1. orig.cpp

```
#include <iostream>

using namespace std;

struct Node
{
    Node *p_next;
    int value;
};

Node* addNode (Node* p_list, int value)
{
    Node *p_new_node = new Node;
    p_new_node->value = value;
    p_new_node->p_next = p_list;

    return p_new_node;
}

void printList (const Node* p_list)
{
    const Node* p_cur_node = p_list;
    while ( p_cur_node != NULL )
    {
        cout << p_cur_node->value << endl;
        p_cur_node = p_cur_node->p_next;
    }
}
```

```
int main ()
{
    Node *p_list = NULL;
    for ( int i = 0; i < 10; ++i )
    {
        int value;
        cout << "Enter value for list node: ";
        cin >> value;
        p_list = addNode(p_list, value);
    }
    printList(p_list);
}
```

示例代码52：orig.cpp

首先，我们来把声明和定义分开。简单起见，我只把真正声明的部分列在下面，其余部分没有变化。

2. orig.cpp

```
struct Node
{
    Node *p_next;
    int value;
};

Node* addNode (Node* p_list, int value);
void printList (const Node* p_list);
```

因为这里不存在文件独有的声明，我们也就不需要做把它们分出去的工作；可以立刻把这些声明都放到一个新的头文件Shared.h中（或者，就这个例子而言，把这个头文件叫做linkedlist.h）。我将完整地列出每个文件。

3. linkedlist.h

```
struct Node
{
    Node *p_next;
    int value;
};

Node* addNode (Node* p_list, int value);
void printList (const Node* p_list);
```

示例代码53：linkedlist.h

4. linkedlist.cpp

```
#include <iostream>
#include "linkedlist.h"

using namespace std;
```

```
Node* addNode (Node* p_list, int value)
{
    Node *p_new_node = new Node;
    p_new_node->value = value;
    p_new_node->p_next = p_list;

    return p_new_node;
}

void printList (const Node* p_list)
{
    const Node* p_cur_node = p_list;
    while (p_cur_node != NULL)
    {
        cout << p_cur_node->value << endl;
        p_cur_node = p_cur_node->p_next;
    }
}
```

示例代码54：linkedlist.cpp

5. orig.cpp

```
#include <iostream>
#include "linkedlist.h"

using namespace std;

int main ()
{
    Node *p_list = NULL;
    for ( int i = 0; i < 10; ++i )
    {
        int value;
        cout << "Enter value for list node: ";
        cin >> value;
        p_list = addNode(p_list, value);
    }
    printList(p_list);
}
```

示例代码55：orig_new.cpp

要注意头文件不应该含有任何函数的定义，如果我们在头文件中加了函数的定义，然后将这个头文件包含进多个源文件中，那么该函数的定义在链接的时候就会出现两次。这会使链接器感到混乱并且会带来不好的后果。

我们同样需要确保函数的声明在同一个源文件中不能重复出现。Orig.cpp可能会引入更多的头文件，这些头文件中有些可能也引入了linkedlist.h：

6. newheader.h

```
#include "linkedlist.h"
// 其他代码
```

7. orig.cpp

```
#include "linkedlist.h"
#include "newheader.h"

/* orig.cpp中的其他代码 */
```

orig.cpp引入了两次linkedlist.h，一次直接引入，一次是通过newheader.h的引入而间接的引入。

解决这个问题需要一个*引用防护*（include guard）。引用防护利用C++预处理器来控制是否需要引入一个文件。基本的思想是说：

```
if <我们还没有引入这个文件>
    <标记一下我们已经引入了这个文件>
    <引入这个文件>
```

可以放心地使用这种模式，因为我们永远都不需要多次引入一个文件。

要实现一个引用防护，我们需要使用`#ifndef`这个在之前遇到过的预处理命令。`#ifndef`语句就是说“if not defined”，对下一个`#endif`出现之前的代码块都有作用。

```
#ifndef ORIG_H
//头文件的内容
#endif
```

这段代码的意思是,如果没人定义过`_ORIG_H`，那么往下执行直到遇见`#endif`。这个小技巧就是现在可以这样定义`ORIG_H`：

```
#ifndef ORIG_H
#define ORIG_H
// 头文件的内容

#endif
```

想象一下如果有人把这个头文件引入了两次会发生什么，第一次的时候`ORIG_H`是未定义的，所以`#ifndef`的作用域包含了该文件剩下的部分，包括定义`ORIG_H`的那部分。（当然，它将`ORIG_H`定义为空，但是`ORIG_H`仍然属于被定义过的。）下一次该文件被引入的时候，`#ifndef`是不成立的，这样就没有代码被引入。

你需要为头文件引用防护起唯一的名称，一个很好的办法就是在头文件的后面加上`_H`。这么做应该能够保证你的引用防护是唯一的，并且不会和别人的`#define`值或者引用防护冲突[①]。

① 你可能还需要在`#define`中加入自己的名字或者公司的名称，如果把代码共享出去或者自己使用了很多共享的代码。因为别人可能也创建了一个叫做链表（linked list）的文件。

21.2.5 关于头文件其他要注意的地方

永远不要直接引入a.cpp文件。引入a.cpp文件会导致问题发生，因为编译器会把.cpp文件中的每个函数定义都编译出一个副本放到引入它的那些目标文件中，然后链接器就会看到同一个函数的很多的定义，不能这样。即使非常小心地引入了.cpp文件，你也会享受不到分开编译节省时间的好处。

关于这个规则有一点值得注意，即你应当只有每个函数的一个副本：对于每次构建，你都应只有一个包含`main`函数的源文件。`main`是程序的入口点，所以只应有它的一个版本。

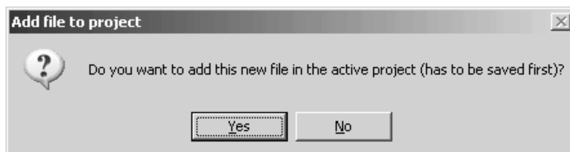
21.2.6 在开发环境中处理多个源文件

如何设置正确的多文件链接取决于开发环境。我会演示一遍各个开发环境里的设置流程，从Code::Blocks开始。

1. Code::Blocks

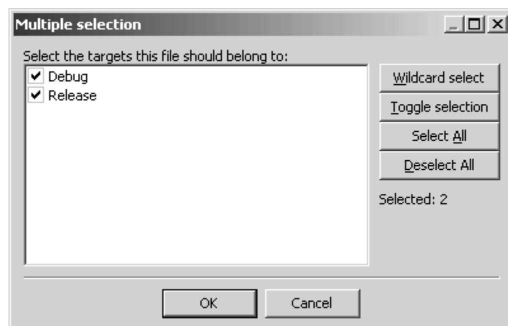
在Code::Blocks中，往项目中添加新的源文件你需要选择File|New|Empty Source File...。

你会被询问是否要将新建的文件加入到当前项目中：



选择Yes。

然后你需要选择一个文件名。当文件名选好之后Code::Blocks会提示让你选择哪个构建配置需要用到这个文件。对于源文件来说，这是真正的将该文件加入链接的步骤。



选择所有的选项（最典型的是Debug和Release）。尽管你不会需要链接头文件，但是新建头

文件时选择这两个选项也是可以的，因为Code::Blocks很智能不会将头文件添加到链接的选项中。

要使用新的文件，你需要同时添加一个头文件和一个源文件，然后把代码改成像之前讲过的那样。

2. g++

如果使用的是g++，你除了在命令行新建一个文件并且给它一个文件名以外不需要做其他特别的事。例如，如果有orig.cpp、shared.cpp两个源文件和一个shared.h头文件，你可以用下面的命令编译这两个源文件：

```
g++ orig.cpp shared.cpp
```

你不需要在命令行里提到头文件，它应当已经被需要它的.cpp文件包含了。这个命令会把给出的文件全部重新编译。如果想要充分利用分开编译的好处，你可以使用-c标识分别编译每个文件：

```
g++ -c orig.cpp
g++ -c shared.cpp
```

然后将它们链接起来：

```
g++ orig.o shared.o
```

或者简单地：

```
g++ *.o
```

这样做的前提是你知道当前目录下不会有什么错误的目标文件。

手动来控制分开编译是件单调乏味的过程。使用makefile来实现它就会简单很多。makefile是你的程序构建过程的描述，它能够为不同的源文件之间的依赖关系编码，这样只要你改动了一个源文件，makefile就能将任何与这个文件有依赖关系的源文件都重新编译。

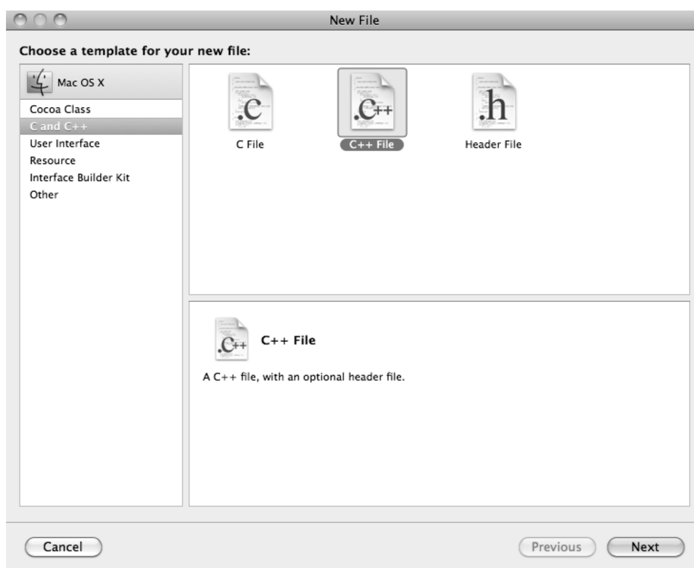
makefile超出了本书所要讲述的范围了，但是你可以在http://www.cprogramming.com/tutorial/makefiles.htm上学习。就算你不打算学习makefile，现在仍然可以使用下面这个命令继续一次编译所有的C++文件：

```
g++ orig.cpp shared.cpp
```

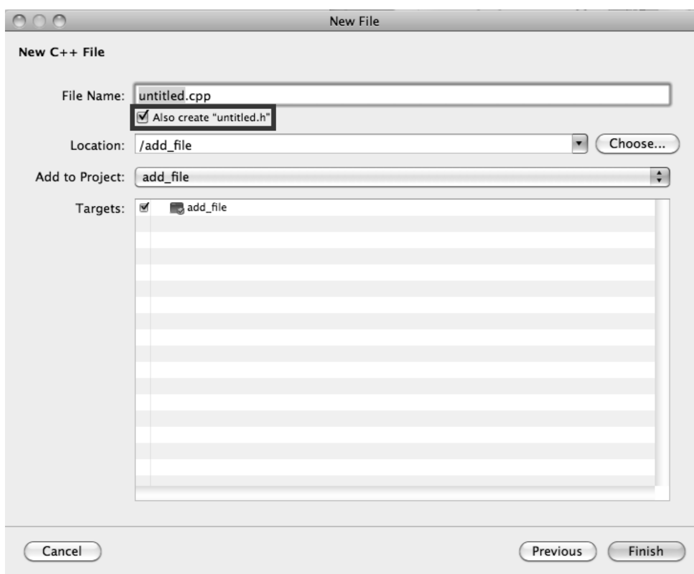
3. XCode

要向XCode项目中添加一个新的源文件可以使用File|New File菜单选项。如果你要确保新文件显示在左边树形视图的Source文件夹下，那就在点开File|New File...之前选择main.cpp所在的Source目录。这么做不是必须的，但是可以帮助你保持一切有序。

选择了File|New之后，你会有几个文件类型的选项：



在左边的面板上选择C and C++，然后在右边选择C++ file（或者，如果只是想要添加一个头文件，那就选择Header file）。如果你想要同时添加一个头文件和一个C++文件，那么就选择C++ file。在下一个界面你将有选择同时创建头文件的选项。单击Next。



填写文件的名字，如果你不愿意将文件放在默认的路径下，可以填一个新的路径。如果你喜欢，接受默认的设置就行了，在这个例子中，我把它直接添加到add_file目录下了，add_file与一个叫做add_file的项目相关联。

如果选择创建C++文件，你可以选择同时创建一个头文件；我在上面的截图中用方框把它框出来了。如果选了它，头文件会在你单击Finish之后自动打开。

XCode会自动设置构建过程来编译你新建的cpp文件并把它和别的文件链接起来。

21.3　问答题

(1) 下列那个不属于C++构建过程中的一部分？

A. 链接　　B. 编译　　C. 预处理　　D. 后续处理

(2) 你在什么时候会遇到一个关于未定义函数的错误？

A. 在链接的过程中　　B. 在编译的过程中
C. 在程序启动时　　D. 在你调用方法的时候

(3) 下列哪项会在你重复引入头文件时发生？

A. 重复声明的错误
B. 没有异常，头文件总是只会被载入一次
C. 与头文件本身是如何实现的有关
D. 头文件一次只能被一个源文件引入，所以不会出现问题

(4) 把编译和链接分开有什么好处？

A. 没有好处，这样做会让人感到迷惑并且有可能很慢因为有多个程序要运行
B. 更容易分析错误因为你可以知道问题出自链接器还是编译器
C. 这样做只让有改动的文件重新编译，节省了编译和链接的时间
D. 这样做只让有改动的文件重新编译，节省了编译时间

21.4　实践题

(1) 写一个程序包含`add`、`subtract`、`multiply`和`divide`的函数。这些函数每个都接收两个整型参数，然后返回相应操作的结果。用这些函数创造一个计算器。把这些函数的声明放到一个头文件中，但是把函数实现的代码直接写在你的源文件中 。

(2) 将上面所写程序中的加减乘除的函数实现从你的计算器程序代码中拿出来，放到一个单独的源文件中。

(3) 找出你在二叉树那一章的练习中实现的二叉树代码，把其中所有的结构体声明和函数声明移到单独的头文件中。将其中的结构体声明放到一个文件中，函数声明放到另一个文件中。将所有的函数实现都放到同一个源文件中。写一个小程序来测试二叉树的基本功能。

第 22 章

程序设计方法介绍

既然已经解决了如何在硬盘上以一种便于大规模程序编写的方式来存储代码，那么我们可以把注意力集中在这个问题的下一步——如何在逻辑上组织代码，让它们便于编辑和处理。我们先看一些最常见的问题，这些问题都是在程序规模越来越大时会遇到的。

22.1 冗余代码

尽管在介绍函数的时候简单地提及了冗余代码的问题，但是我们现在还需要更深入地看看这个问题。随着程序规模越来越大，逻辑会一遍一遍地重复。比如，你在写一个游戏，就需要代码把不同的图形元素绘制到屏幕上（例如，飞船或者子弹）。

在能够绘制飞船之前，你需要最基本的功能来绘制一个像素，一个像素就是屏幕上使用二维坐标来定位的一个有颜色的点。大部分时候，可以借助于图形库来进行这种绘制[①]。

你还需要代码来实现使用这些像素点（或者图形库可以提供的别的基本图形，如线段和圆）绘制出真正的游戏元素——飞船、子弹等。

你可能需要在代码中很频繁地进行这样的绘制——每次飞船或者子弹移动时它们肯定需要被重新绘制。如果每次需要绘制子弹的时候都写一段绘制子弹的代码，那么你就写了很多冗余代码。

这些冗余代码给程序带来了不必要的复杂度，让程序难以让人理解。你需要有标准的方式来做某些事情，像绘制飞船或者子弹，而不是让代码任意的部分都可以去重复这些过程。这样做道理何在呢？我们假设要修改一个东西——也许是子弹的颜色吧。如果在10个不同的地方都有显示子弹的代码，你最后不得不修改每个地方，而这仅仅为了修改一下子弹的颜色。这太痛苦了！

每次要绘制一颗子弹，你得重新写一遍绘制子弹的代码，或者去找一段现成的代码复制粘贴过来，也许还要修改一些变量名来避免冲突。哪种方式你都得想想“如何绘制一颗子弹”而不是

① 本书中不会使用图形，但是你可以了解更多：http://www.cprogramming.com/graphics-programming.html 。

“给我绘制一颗子弹”。此外，在回头看代码时，你还要去弄清楚这段代码是实现什么功能的——相对来说很难弄清楚下面这段代码：

```
circle( 10, 10, 5);
fillCircle(10, 10, RED);
```

是在绘制一颗子弹，而这样表示：

```
displayBullet(10, 10);
```

更容易让人明白这是在绘制一颗子弹。

函数可以赋予代码块有意义的名字，这样在阅读代码时你可以记得这段代码是实现什么功能的。也许你尚未体会到，当构建更大规模的程序时，花在阅读代码上的时间将比写代码的时间还多，所以好的命名和好的函数会产生很大的影响。

22.2　假定数据是如何存储的

冗余的问题不光会影响到算法。我们来看看另外一段有隐藏冗余的示例代码。假使你想要实现一个棋类程序，其中棋盘上的位置用数组来表示如何呢？每次访问棋盘，你可以简单地访问一下数组。

初始化数组的第二列，让它存储所有的白色卒子，你可能这样写：

```
enum ChessPiece { WHITE_PAWN, WHITE_ROOK, /* 其他变量 */ };

// …… 很多代码

for ( int i = 0; i < 8; i++ )
{
    board[i][1] = WHITE_PAWN;
}
```

过后，如果想要查看某个方格上放的是什么棋子，可以直接从数组中读取：

```
// …… 很多代码
if ( board[0][0] == WHITE_ROOK )
{
    /* 运行某些代码 */
}
```

随着程序规模的扩大，越来越多的访问棋盘的代码会乱七八糟到处都是。这有什么害处呢？你每次从数组中读取数据时并不是在做重复的事，仅仅需要一行代码，对吧？但是，你又确实在做重复的事——在重复使用同样的数据结构。重复地使用相同的数据结构时，你的代码其实是已经假设了棋盘是如何存储的。你不是在重复算法逻辑，而是在重复数据是如何存储的这一假设。这么来思考吧，这里因为凑巧只需要一行代码来访问棋盘，这并不意味着在什么时候访问棋盘都

是只需要一行代码。如果你以不同的方式来实现棋盘，也许需要更复杂的技巧来访问棋盘。

复杂的棋类程序使用一种不同于数组的方式来表示棋盘。（它们使用多位板[1]而不是数组，这些位板每次访问的时候都需要不止一行代码。）如果要写一个棋类程序，我可能开始时先使用数组，这样可以专注于基本的算法，然后再去考虑代码优化得更快的问题。但是为了更方便地改变棋盘的存储，我会把数组隐藏起来。可是，如何隐藏数组呢？

上一次需要隐藏某些实现逻辑时，我们是想要隐藏绘制子弹的细节。我们是通过使用一个可以调用的函数，而不是直接写出绘制子弹到屏幕上的代码来实现的。这里同样可以使用一个函数来隐藏棋盘存储的细节。不直接访问数组，而是调用一个访问数组的函数。例如，你可以写一个像下面这个getPiece一样的函数：

```
int getPiece (int x, int y)
{
    return board[x][y];
}
```

22

我们发现上面的函数需要两个参数，然后它返回一个数值，就像访问数组一样。这样做并没有让你少写代码，因为需要传入的参数和之前一样——一个x坐标和一个y坐标。所不同的是访问棋盘的方式现在被隐藏在这个函数中了。你其余的代码中可以（并且应该）调用这个函数来访问数组。然后，如果你决定改变棋盘的存储方式，可以仅仅修改这个函数——其他的地方不受影响[2]。

使用函数来隐藏细节的思想有时称为*函数抽象*。应用函数抽象意味着你应当把任何重复的操作放到一个函数中——让这个函数为调用者指定输入和输出，但避免让调用者知道这个函数是如何实现的。这里的如何实现可以是使用的算法，或者是使用的数据结构。该函数允许它的调用者利用它所提供的接口的可靠性承诺，从而不需要知道这个函数是如何实现的。

这里有一些使用函数来隐藏数据和算法的好处。

(1) 让以后的工作更加轻松。你只需要使用一个之前写的函数就行了，而不是一直记着怎样实现算法逻辑。只要你相信该函数对于合法的输入都能正常工作，就可以信任它的输出而不需要记得它是如何工作的。

(2) 一旦你能够信任某个函数“可以工作”，就可以开始一遍遍地使用它来写代码解决问题。你无需担心任何细节（像如何访问棋盘），这样就可以专注于解决新的问题（比如如何实现AI）。

(3) 如果发现逻辑中有个错误，你不需要修改代码中的很多地方，只需要修改一个函数而已。

---

① 参考http://en.wikipedia.org/wiki/Bitboard 。

② 前提是，你一直调用这个方法来访问棋盘。你可能还需要几个在棋盘上设置棋子的函数，但是修改两个函数终归比修改几十个几百个好多了。

(4) 如果通过函数来隐藏数据结构，你同样也会增强自己存储和表现数据的灵活性。你可以先用效率不高但是便于编写的方式，然后如果有需要的话，再把它替换成更快速高效的实现方式，完成这些只需要修改少数几个函数，别的都不用动。

22.3　设计和注释

在写精心设计的函数同时，你还应该给它们注释。虽然给函数添加注释不是听起来那么简单。

好的注释可以解答读者的疑问。

本书示例中你看到的那些注释——像这个：

```
// 声明变量i并初始化为3
int i = 3;
```

可不是真正需要写的注释！这样的注释只是为了回答编程初学者的疑问；但是在现实环境中，阅读你代码的人是已经了解了C++的。

还有更糟糕的情况，随着时间的推移注释过期了。如果有人读了这样的注释，不光浪费了他们的时间，还可能让他们完全误解了代码的意义。

写一些表达疑问的注释会好很多，比如“啊，这貌似是个奇怪的方式。他们为什么这么做呢”，或者“这个函数可以接受哪些参数值，它们又代表什么意思呢”。下面是个注释的示例，你应当努力为所写的函数加上这样的注释：

```
/*
 * 根据给定的正整数n计算斐波那契数列值。如果n的值小于1，
 * 该函数返回1
*/
int fibonacci (int n);
```

我们发现上面函数的描述准确表达了该函数的功能，哪些参数是合法的，并且遇到非法参数时会发生什么情况。这种注释表示使用该函数的人无需再去看它是如何实现的，这很好！

好的注释并不是啰嗦的注释——你不应该每一行代码都加注释。我通常给那些为了在文件以外调用的函数添加注释，并且我会给特别绕人或者看起来怪异的代码添加解释性的注释。

有一个过分精简注释的坏习惯，那就是在开发周期的最后再来添加注释。一旦编码都已完成，再去回顾并且添加有意义的注释就显得太晚了；你所做的只是添加你在阅读代码时所能了解到的信息。在写代码的同时就添加的注释是最有用的。

22.4 问答题

(1) 使用函数而不直接访问数据的好处是什么?

A. 函数可以被编译器优化来提供更快的访问速度
B. 函数可以对调用者隐藏自己的实现逻辑，这样便于改变该函数的调用者
C. 使用函数是在多个源文件之间共享同一个数据结构的唯一途径
D. 没有什么好处

(2) 在什么情况下应该把代码放进一个通用的函数中呢?

A. 在你需要调用它的时候
B. 在你开始从很多地方调用同一段代码的时候
C. 在编译器开始抱怨函数太大而不能编译的时候
D. B和C

(3) 为什么要隐藏数据结构的表示方式?

A. 让数据结构更便于替换
B. 让使用该数据结构的代码更容易让人理解
C. 让代码中别的地方使用该数据结构时更容易
D. 以上都正确

第 23 章

隐藏结构化数据的表示

到目前为止，你已经看到如何隐藏存储在全局变量或者数组中的数据。隐藏数据并不局限于这几个例子。创建结构体时往往是你最想隐藏数据的时候之一。这可能让你觉得奇怪：毕竟一个结构体有一个非常特殊的布局和可以存储的一系列数值。当你以一组字段的方式看待它们时，结构体无法提供隐藏实现细节的方式（例如它们以何种形式存储哪些字段）。实际上，你可能觉得奇怪："难道一个结构体的全部意义不是为了提供一些特定的数据吗？为什么要隐藏这些数据的表示呢？"事实证明，还可以用另外一种方式来思考结构体，在这种方式下的确需要隐藏数据。

大部分时候，当有一堆相关的数据，真正重要的并不是你如何存储这些数据而是用这些数据做什么。这一点非常重要，它可以成为一个观念变革。所以我将再重复一遍：*真正重要的并不是如何存储数据，而是如何使用数据*。

由于粗体文本并不总是能够一看就明了，让我们举一个简单例子——字符串。除非你真正自己实现字符串类，否则无所谓怎么存储字符串。对于任何一段运用字符串的代码，重要的是如何得到字符串的长度、访问单个字符或者显示字符串。字符串的实现可能使用一个字符数组，然后用另一个变量来存储长度，也可以使用一个链表，或者使用一个你从来没听说过的C++的特性。

作为字符串的使用者，无所谓字符串是怎么实现的——重要的是可以用字符串做什么。你可以做许多事，但就算是C++字符串也只可以做约35种操作——而且大部分时候它们中的大部分操作都是用不到的。

你将经常需要的是在不暴露实现某个数据类型的原始数据的基础上创建新的该数据类型的能力。例如，当创建一个字符串时，你不需要担心保存字符的缓冲区。STL向量和映射正是这样工作的；你不需要为了使用它们而去了解它们的实现方式——所要注意的是，当使用一个STL向量时，它的实现可能是像用胡萝卜来喂食超活跃的兔子，同时注意组织上的小技巧。

使用函数来隐藏结构的布局

你可以通过创建与结构体相关联的函数来隐藏具体的字段。例如，想象一个小棋盘代表局势

和双方的移动（白色或黑色）。我们将使用枚举类型来存储棋子和将要走棋的玩家：

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* 其他变量 */ };
enum PlayerColor { PC_WHITE, PC_BLACK };

struct ChessBoard
{
    ChessPiece board[ 8 ][ 8 ];
    PlayerColor whose_move;
};
```

你可以创建操作棋盘的函数，把棋盘作为该函数的参数：

```
ChessPiece getPiece (const ChessBoard *p_board, int x, int y)
{
    return p_board->board[ x ][ y ];
}

PlayerColor getMove (const ChessBoard *p_board)
{
    return p_board->whose_move;
}

void makeMove (ChessBoard* p_board, int from_x, int from_y, int to_x, int
to_y)
{
    // 通常情况下，我们首先需要写点代码验证移动棋子的合法性
    p_board->board[to_x][to_y] = p_board->board[from_x][from_y];
    p_board->board[from_x][from_y] = EMPTY_SQUARE;
}
```

你可以把它们当做其他任何一个函数一样使用：

```
ChessBoard b;
// 首先需要初始化棋盘

// 接下来就可以像下面这样使用它了
getMove( & b );

makeMove( & b, 0, 0, 1, 0 ); // 把一个棋子从0，0 移动到1，0
```

这是一个好方式，事实上，C语言程序员使用这种方式已经很多年了。另一方面，这些函数只与ChessBoard结构体相关联，因为它们正好把ChessBoard作为一个参数。没有地方明确地表示："这个函数应该被当做该结构体的核心部分。"一个结构体不仅包含数据，而且包含了操纵数据的函数，这么说不是很好吗？

C++认真考虑了这个想法并且直接把它构建到了语言中。为了支持这种风格，C++引入了方法的概念——方法就是作为某个结构体的一部分来声明的函数（在之前关于STL的部分我们接触过方法）。不像不受约束的函数和结构体没有什么关联，方法可以很简单地操作存储在结构中的数据。方法的作者把方法作为结构体的一部分来声明，这样就直接把方法与结构体联系在了一起。

声明了结构体的方法部分以后，方法的调用者就不需要把该结构体作为一个单独的参数了！虽然这需要特殊的语法 。

方法声明和调用的语法

来看看如果把函数变成方法会怎么样：

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* 及其他 */ };
enum PlayerColor { PC_WHITE, PC_BLACK };
struct ChessBoard
{
    ChessPiece board[ 8 ][ 8 ];
    PlayerColor whose_move;
    ChessPiece getPiece (int x, int y)
    {
        return board[ x ][ y ];
    }
    PlayerColor getMove ()
    {
        return whose_move;
    }
    void  makeMove (int from_x, int from_y, int to_x, int to_y)
    {
        // 通常情况下，我们首先需要写点代码验证移动棋子的合法性
        board[ to_x ][ to_y ] = board[ from_x ][ from_y ];
        board[ from_x ][ from_y ] = EMPTY_SQUARE;
    }
};
```

示例代码56：method.cpp

首先可以看到，方法是在结构体里面声明的。这很明显，这些方法应被作为该结构体的基本组成部分来看待。

此外，这些方法声明不需要单独接收一个ChessBoard类型的参数——在方法里面，结构体所有的字段都可以直接使用。写下board[x][y]就可以直接访问该方法所在结构体的棋盘。可是代码怎么知道它所使用的方法属于哪个结构体的实例呢？（如果有不止一个ChessBoard怎么办？）

像下面这样调用一个方法：

```
ChessBoard b;
// 初始化棋盘的代码
b.getMove();
```

调用与某个结构体相关联的函数时看上去和访问该结构体的字段几乎是一样的。

在内部，是编译器在处理如何让方法访问它所在结构体中的数据的细节。从概念上讲，< variable >.< method >的语法是将< variable >传递给< method >的简写形式。现在

你明白了为什么在讲STL那一章中我们需要这个语法了吧，那些函数就像这些方法一样运作。

把方法的定义从结构体中移出来

把所有的函数体都包含在结构体中真的会很乱而且让人难以理解。所幸，你可以把方法拆分成一个在结构体中的声明和一个放在结构体之外的定义。例子如下：

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* 及其他 */ };
enum PlayerColor { PC_WHITE, PC_BLACK };
struct ChessBoard
{
    ChessPiece board[ 8 ][ 8 ];
    PlayerColor whose_move;

    // 在结构体中声明方法
    ChessPiece getPiece (int x, int y);
    PlayerColor getMove ();
    void makeMove (int from_x, int from_y, int to_x, int to_y);
};
```

现在方法的声明在结构体内部了，但是其他方面看上去像普通函数的原型。

方法的定义需要一些方式回头来把它们自身与结构体联系起来——我们可以使用一个特殊的“范围”语法来表示该方法是属于某个结构体的。这个语法就是像`<structure name>::<method name>`这样来写方法的名字，但是从其他方面来看代码没有变化：

```
ChessPiece ChessBoard::getPiece (int x, int y)
{
    return board[ x ][ y ];
}

PlayerColor ChessBoard::getMove ()
{
    return whose_move;
}

void  ChessBoard::makeMove (int from_x, int from_y, int to_x, int to_y)
{
    // 通常情况下，首先需要写点代码验证移动棋子的合法性
    board[ to_x ][ to_y ] = board[ from_x ][ from_y ];
    board[ from_x ][ from_y ] = EMPTY_SQUARE;
}
```

本书的后面部分，我会把行数稍微多一点的方法的声明和定义分开。有些业内人士建议永远不要在结构体内部定义方法因为这样会暴露方法是如何实现的，而这是不必要的。你暴露的方法实现越多，就越可能有人依赖方法的具体实现细节来写代码而不是仅仅依靠方法的接口。本书中，我有时会把方法声明放在类中就是为了节省点空间。

23.1 问答题

(1) 你为什么需要使用方法而不是直接使用结构体的字段？

A. 因为方法更加易读
B. 因为使用方法程序会更快
C. 你不应该使用方法，就应该直接使用字段
D. 这样做你可以修改数据的表现形式

(2) 下列哪个定义了与结构体`struct MyStruct { int func(); };`相关联的方法？

A. `int func() { return 1; }`
B. `MyStruct::int func() { return 1; }`
C. `int MyStruct::func(){return 1;}`
D. `int MyStruct func () { return 1; }`

(3) 你为什么想要把方法的定义内联在一个类中？

A. 这样可以让该类的使用者看到这个方法是怎么工作的
B. 因为这样会让代码跑得更快
C. 你不能这么做！这样会泄漏方法实现的细节
D. 你不能这么做，这会让程序跑得更慢

23.2 实践题

写一个结构体为井字棋棋盘提供接口。用基于该结构体的方法来实现一个双人对战的井字棋。要求像走棋和检测是否某个玩家胜利这样的基本操作都属于该结构体的接口。

第 24 章 类

Bjarne Stroustrup在创造C++的时候，真正想强化的是由方法来定义结构体的思想，而不是实现结构体时碰巧用到的那些数据。他本来可以通过扩展已有结构体的概念来实现他想要的，但是他没有，相反他创造了一个新的概念：类。

类就如同一个结构体，只不过它能够定义哪些方法和数据是属于类内部，哪些方法是为了提供给该类的使用者的。你应当把类的意思想作和种类一样，定义一个类的时候就是在创造一个新类别的东西或者说新种类的东西。它不再具有作为结构化数据的内涵性，相反，类是由那些它作为接口向外部提供的方法来定义的。类甚至能够防止你不小心使用其具体的实现细节。

是这样的——在C++中，阻止不属于某个类的方法使用该类的内部数据是可以实现的。实际上，当你声明一个类的时候，默认情况就是除了该类自身的那些方法以外，没有人能够使用该类的任何内容！你得明确地表示哪些内容可以被公共访问。使数据在类以外不可访问的功能可以让编译器检查程序员没有在使用那些他们不该碰的数据。这对于程序的可维护性来说可谓是神来之笔。你可以修改类的一些基本的东西，比如棋盘的存储方式，而不用担心这样会破坏类以外的代码。

就算项目只有你一个人在做，保证没有人能“作弊”以及看到方法的内部实现，实际上也是一件美事。其实，说方法很有用还有另外一个原因，你很快就会看到的，只有方法才能访问“内部”数据。

从这里往后，在我想要隐藏数据存储方式的时候我都会使用类，在绝对没理由隐藏的时候我会使用结构体。你可能会惊讶于结构体用的有多稀少——数据隐藏就是这么有价值。在实现类并且需要一个辅助性的结构体来存放部分数据时，是唯一要使用结构体的时候。由于辅助性的结构体仅仅是针对这一个类的，并且不需要公开暴露，所以通常没有必要把它写成一个完整的类。如我所说，没有硬性的需求一定要这样做，但是这么做是约定俗成的。

24.1 隐藏数据的存储方式

我们来研究一下类里面隐藏数据的语法——你如何使用一个类来隐藏一些数据同时把一些方法提供给所有人呢？类可以让你把每个方法和字段（通常被称为类的成员）归结为公共或者私

有——公共成员所有人都可以访问，私有成员只有该类中其他的成员可以访问[①]。

下面是个例子，将方法都声明为公共的，而所有的数据都声明成私有的：

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* 及其他 */ };
enum PlayerColor { PC_WHITE, PC_BLACK };

class ChessBoard
{
public:

    ChessPiece getPiece (int x, int y);
    PlayerColor getMove ();
    void makeMove (int from_x, int from_y, int to_x, int to_y);

private:
    ChessPiece _board[ 8 ][ 8 ];
    PlayerColor _whose_move;
};

// 方法的定义和之前完全相同!
ChessPiece ChessBoard::getPiece (int x, int y)
{
    return _board[ x ][ y ];
}

PlayerColor ChessBoard::getMove ()
{
    return _whose_move;
}

void  ChessBoard::makeMove (int from_x, int from_y, int to_x, int to_y)
{
    //通常情况下，首先需要写点代码验证移动棋子的合法性
    _board[ to_x ][ to_y ] = _board[ from_x ][ from_y ];
    _board[ from_x ][ from_y ] = EMPTY_SQUARE;
}
```

示例代码57：class.cpp

我们发现这个类的声明和之前结构体的声明看上去很像，除了一个主要的区别。我使用了两个新的关键字：public和private。任何在public关键字之后声明的东西，所有人都可以通过该类的对象来使用（在这里就是getPiece、getMove 和makeMove这些方法）。任何出现在private之后的东西，都只能被ChessBoard类自身的方法访问到（_board和_whose_move）[②]。

① 还有第三种类型，叫做protected，我们稍后会讨论到。

② 我还在类的每个私有元素之前加了下划线，以便分别出哪些是私有的，但是这不是C++的要求。这么做一开始看上去有点丑陋，但是在阅读代码的时候你会发现它作用可大了！如果你要遵守这个习惯，就要确保在下划线后面没有紧跟着一个大写字母；那样做可能会在编译器那里产生冲突。只要你保证声明私有变量或者方法时在下划线后面跟一个小写字母，就不会出乱子。

顺便说一下，你可以随意调换public和private的位置。下面的这个类和前面的那个类声明了相同的公共内容：

```
class ChessBoard
{
public:

    ChessPiece getPiece (int x, int y);

private:
    ChessPiece _board[ 8 ][ 8 ];
    PlayerColor _whose_move;

public:
    int getMove ();
    void makeMove (int from_x, int from_y, to_x, to_y);
};
```

我自己写代码的时候，总是先以一个public区块开始，跟着来个private区块。这么做是在强调public区块是为了使用这个类的人而写的（也就是别的程序员），因为它会是使用这个类的人首先会看到的东西[①]。

24.2 声明一个类的实例

声明一个类的实例就如同声明一个结构体的实例一样：

```
ChessBoard b;
```

在类上进行方法的调用也是和结构体的一模一样：

```
b.getMove();
```

虽然有一个小的术语上的差别。你声明某个类的一个变量时，那个变量通常被称为对象。对象这个词应当代表现实世界中事物的抽象，比如方向盘——这种暴露一个很小的接口而后面隐藏了很多复杂的东西。当你要把汽车往左转的时候，只需要打方向盘——不必担心那些齿轮是怎么工作的。你所要做的就是转动方向盘并且踩油门。所有的细节都被隐藏在一个基本的用户界面之后。在C++中，一个对象所有的实现细节都被隐藏在一系列公共方法的调用之后——这些方法就是组成类的“用户接口”的东西。一旦你定义了一个接口，类可以随意地去实现它——怎么存储数据以及方法如何去实现，都由你来决定。

① 这些用户当然是指别的程序员，而不是软件的用户。很多情况下，你将会是自己所写类的用户。

24.3　类的职责

在你创建一个C++类的时候，把它想作创建了一个新型变量——一个新的数据类型。你的新数据类型就如同一个整型或者一个字符串，但是功能更强大。你已经看到过这种思想——在C++中，字符串是一个类，实际上，字符串类是你可以使用的一个新的数据类型。公共和私有的思想在你想要创建新的数据类型时非常有意义：你是想要为外部提供一些特定的功能和一个特定的接口。举个例子，一个字符串提供了显示自己，处理子字符串或者单个的字符，以及获取字符串长度这样的基本属性等功能。字符串自身是如何实现真的无关紧要了。

如果把创建一个类想作是在定义一个新的类型，那么首先需要做的就是弄清哪些需要设为公共的：你想要类做哪些事。公共的任何东西都可以被使用这个类的人所用——你应当把它作为接口来对待，就像一个函数，有一个接口包含了所要接收的参数和返回值。这是你需要仔细思考的东西，因为一旦开始使用这个接口，再去改变这个接口的话就需要同时修改所有的这个接口的使用者。由于方法是公共的，就会有很多很多的调用者——你无法找到一个轻松的方式来限制接口被调用的次数。没有人会发明一个全新的开车方法因为这样的话所有人都要重新学习一遍怎么开车！但是发明一个新型的引擎是完全可以的，比如从纯汽油过渡到混合动力，因为这并没有改变接口，这改变的只是具体实现。

一旦你提出了一个公共接口，就应该开始思考如何去实现组成接口的这些公共方法。任何用来实现公共方法的方法或者字段，如果不需要设为`public`就应该设为`private`。

与公共接口相反，私有方法和数据是很便于修改的。只有该类的方法可以使用类的这些私有成员（公共方法和私有方法都可以）。把实现细节设为私有，在以后如果决定要重新实现类的功能，你就有机会修改它们。（第一次就把它都写对是很困难的）记住混合动力的汽车就是个例子！

我的建议很简单：永远不要把数据字段设为`public`，将方法默认都设为`private`，如果你确信哪些方法应该设为`pubic`，那么再把它们移到公共接口中从`private`到`public`简单，从`public`到`private`很难——正所谓覆水难收。如果你需要为某个特定的字段提供访问接口，那么就写一些方法来获取以及设置它们的数值：如果它们是用来读取变量的，那么这些方法通常称为*获取方法*（getter），如果是用来写入变量的，则称为*设置方法*（setter）。

从不把字段设为`public`的做法有时看上去有点迂腐。你不是得写很多获取方法和设置方法，写很多像`getMove`这样的函数什么都不做只是返回一个如`_whose_move`这样的私有字段吗？

不错，有时确实是这么回事。在你意识到需要修改一个不起眼的获取方法，来添加某种功能却发现自己陷入困境的时候，写这些方法消耗的少量精力就不值一提了。举个例子，你可能会决定从把一个值存储在变量中修改为通过其他的一些变量来计算出这个值。如果没写获取方法，而是让所有人都是以`public`字段的形式来访问该数据，这时你就傻眼了。

可能你会想出一些例子，其中有些字段可以很安全地设为`public`。但是我的建议是不要去尝试——在前期为自己省下一点敲键盘的时间，但是却给以后埋藏了一个潜在的大问题，而且尝试这种错误的后果是产生一个糟糕设计，你还无法轻易地修改它。

private 真正的意义是什么

某个东西被声明成`private`并不意味着就有了全面的安全保障。一个类的私有字段都被存储在内存中，就像公共字段一样，通常私有字段紧挨着公共字段；任何代码都可以用神奇指针的把戏来读取这些数据。操作系统和编程语言不会为保护私有数据免受恶意的第三方攻击做出任何保证。把数据设为私有可以让编译器阻止对于私有数据的意外使用——不是为了增强安全保障。虽然这么做没有提供安全保障，但是仍然很有用。

顺便说一下，有一个广泛使用的编程术语来形容使用公共方法来隐藏私有数据：封装。封装意味着隐藏你的实现（封装它），这样使用类的人只需要处理构成类的接口的那一系列方法就行了。也许使用像“数据隐藏”或者“实现细节”的词组来形容更形象一点，但是封装是你会时常遇到的术语。现在你已经知道它是什么意思了。

24.4 小结

类是现实中大部分C++程序的一个基本的组成部分。类可以让程序员创建易于理解和操作的大规模的设计。现在你已经学过了类的一个强大特性——隐藏数据的能力——接下来的几章会介绍更多类的特性。

24.5 问答题

(1) 为什么要使用私有数据?

A. 为了让数据更安全，免受黑客攻击
B. 为了防止其他程序员接触那些数据
C. 为了分清楚哪些数据是应该只用来实现类的
D. 你不应使用私有数据，那样会使程序更难写

(2) 类和结构体有什么不同?

A. 没什么不同
B. 类默认所有成员都是公共的
C. 类默认所有成员都是私有的
D. 类可以让你指定字段是公共的还是私有的，结构体不能

(3) 你应该怎样处理类当中的数据字段?

A. 把它们默认设为公共的
B. 把它们默认设为私有，如果有需要就移到公共的部分
C. 永远不要把它们设为公共的
D. 类通常都没有数据，但是如果有，直接使用

(4) 你如何决定一个方法是否应该设为公共的?

A. 永远不要把方法设为公共的
B. 一直把方法设为公共的
C. 如果方法需要使用类的主要特性就把它设为公共的，否则设为私有的
D. 如果有人可能会想要使用这个方法，那么就把它设为公共的

24.6　实践题

把上一章结尾实践题中的结构体（表示一个井字棋盘的）拿出来并且用类来重新实现它，把有公共作用的方法设为公共的，把数据和辅助性的方法设为私有的。再看看你需要修改多少代码?

第 25 章

类的生命周期

25

创建一个类的时候，你会想让它尽可能地易于使用。有三个基本的操作可能所有的类都需要支持：

(1) 初始化自己；

(2) 清理占用的内存或者别的资源；

(3) 复制自己。

这三点对于创建一个好的数据类型来说都很重要。拿字符串来做个例子：字符串需要能够初始化自身，哪怕初始化成一个空字符串。这个操作不应该依赖某些外部代码来完成——只要你声明了一个字符串，它立刻就可以为你所用。而且，在你用完字符串之后，它需要自我清理，因为字符串是分配过内存的。使用字符串时，你并不需要调用一个方法来做清理的工作；清理是自动搞定的。最后，允许变量之间相互复制也是需要的，就像一个整型数据可以从一个变量复制到另一个变量一样。综上所述，这三个功能应当成为每个类的组成部分，这样的话这些类就很容易被正确地使用并且不易被误用。

我们一个个来分析这三个特性，从初始化对象开始，看看C++是如何让初始化很简单地实现的。

25.1 对象构造

可能之前你就注意到在`ChessBoard`的接口（类的公共部分）中并没有初始化棋盘的代码。来修正一下这个问题。

声明一个类的变量时，需要有一些初始化这个变量的方式：

```
ChessBoard board;
```

在C++中，在一个对象被声明时运行的代码称为*构造函数*。构造函数中应该会设置好相应的对象，这样在使用这个对象的时候就不需要再做进一步的初始化了。构造函数也可以接收参数，在声明特定大小的vector时你已经见识过了。

```
vector<int> v( 10 );
```

这行代码带着参数10去调用vector的构造函数；vector的构造函数初始化一个新的vector这样它就立即可以存放10个整数。

要创建一个构造函数，你只需简单地声明一个和类有着同样名字的方法，不接受参数也没有返回值。（返回值也不是void——字面上你都不需要为返回值指定一个类型。）

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* and others */ };
enum PlayerColor { PC_WHITE, PC_BLACK };

class ChessBoard
{
public:

    ChessBoard (); // <--无返回值
    PlayerColor getMove ();
    ChessPiece getPiece (int x, int y);
    void makeMove (int from_x, int from_y, int to_x, int to_y);

private:
    ChessPiece _board[8][8];
    PlayerColor _whose_move;
};

ChessBoard::ChessBoard () // <-- 仍然没有返回值
{
    _whose_move = PC_WHITE;
    // 先把整个棋盘清空，然后再填入棋子
    for ( int i = 0; i < 8; i++ )
    {
        for (int j = 0; j < 8; j++ )
        {
            _board[ i ][ j ] = EMPTY_SQUARE;
        }
    }
    // 其他初始化棋盘的代码
}
```

示例代码58：constructor.cpp

（如果方法没有改变的话，我不会把它们所有的定义都写在这里，但会一直给你看完整的类声明，这样可以看到它们是如何整合在一起的。）

注意，构造函数是属于类当中公共区域的一部分。如果ChessBoard构造函数不是公共的，那么就无法创建出该对象的实例。何以如此呢？每次创建对象的时候都会调用到构造函数，但是如果它是私有的，那就意味着类之外没有人能够调用到这个构造函数！由于所有的对象都必须调用构造函数来初始化，如果构造函数是私有的你根本就无法声明对象了。

调用构造函数的地方正是创建对象的那行代码：

```
ChessBoard board; // 调用ChessBoard的构造函数
```

或者在分配内存的地方：

```
ChessBoard *board = new board; // 调用ChessBoard的构造函数，分配内存
```

如果你声明了多个对象：

```
ChessBoard a;
ChessBoard b;
```

构造函数的运行顺序和对象声明顺序一致（先a后b）。

就像普通函数一样，构造函数可以接收任意数量的参数，并且你也可以有多个参数类型不同的重载构造函数，如果想要对象可以用不同的方式来初始化的话。举个例子，你可以再写个ChessBoard的构造函数，接收棋盘的大小作为参数：

```
Class ChessBoard
{
    ChessBoard ();
    ChessBoard (int board_size);
};
```

构造函数的定义和类当中其他任何方法一样：

```
ChessBoard::ChessBoard (int size)
{
    // ……代码
}
```

像下面这样通过构造函数来传递参数：

```
ChessBoard board( 8 ); // 8是传递给ChessBoard构造函数的一个参数
```

当使用new关键字时，参数的传递就像你直接调用构造函数一样：

```
ChessBoard *p_board = new ChessBoard( 8 );
```

语法上有个小的注意点——尽管你是使用括号来将参数传递给构造函数的，但是在声明一个构造函数不接受参数的对象时可不能还使用括号。

错误代码

```
ChessBoard board();
```

上面代码正确的写法是：

```
ChessBoard board;
```

然而，在使用new来创建对象时使用括号是没有问题的：

```
ChessBoard *board = new board();
```

上面的这种情况是由于C++解析时的一个不好的怪招导致的（个中细节太过晦涩难懂）。在声明一个没有传参构造函数的对象时要避免使用括号。

25.1.1　没有新建构造函数的结果

如果你没有写构造函数，那么C++就会很友好地创造一个。自动创造的这个构造函数不接收参数，但是它会调用你类中所有字段的默认构造函数来初始化它们（虽然它不会初始化原始类型如整型或者字符串——所以要留心这一点）。我通常会建议写自己的构造函数，以确保所有的东西都按你的意愿来初始化。

一旦为类声明了一个构造函数，C++就再也不会为你自动生成默认的构造函数了——编译器就会假定你知道自己在做什么，并且假定是想要为这个类创建所有的构造函数。尤其是，如果创建了一个接收参数的构造函数，代码就再也不会有一个默认的构造函数，除非你特地声明一个。

这会产生吓人的后果。如果代码先前是使用自动生成的默认构造函数，然后你添加了一个自己的、接收一个或者更多参数的非默认构造函数，依赖之前自动生成的默认构造函数的代码将再也无法编译。你不得不手动地提供一个默认构造函数，因为编译器不再为你创造了。

25.1.2　初始化类的成员

类的每一个成员都需要在构造函数中来完成初始化。假设有个字符串作为ChessBoard类的一个成员：

```
class ChessBoard
{
public:

    ChessBoard ();

    string getMove ();
    ChessPiece getPiece (int x, int y);
    void makeMove (int from_x, int from_y, int to_x, int to_y);

private:
    PlayerColor _board[8][8];
    string _whose_move;
};
```

当然可以简单地给_whose_move变量赋值：

```
ChessBoard::ChessBoard ()
{
    _whose_move = "white";
}
```

尽管真正在这里执行的代码可能有点出乎我们的意料。首先，在ChessBoard构造函数刚开始的时候，_whose_move的构造函数将会被调用。这样是有好处的因为它意味着在构造函数中你可以安全地使用类当中任何的字段——如果那些成员的构造函数不被调用，它们就无法使用——构造函数的全部意义就是让对象可以使用！

可以给类成员的构造函数传参，如果你打算这么做，而不是直接使用默认构造函数的话。尽管这个操作的语法有点不同寻常，但是它是有效的：

```
ChessBoard::ChessBoard ()
    // 跟在冒号后面的是变量的列表，带着传递给构造函数的参数
    : _whose_move( "white" )
{
    // 代码运行到这里的时候，_whose_move 的构造函数已经被调用了
    // 并且它已经有了值"white"
}
```

上面语法的术语叫做*初始化列表*。后面会有几次遇到它们，并且我通常都会用这个语法来初始化类的成员。初始化列表的成员之间使用逗号分隔开。举个例子，如果给ChessBoard增加一个新的成员来计算已经走过的步数，可以像这样在初始化列表中对它进行初始化：

```
class ChessBoard
{
public:

    ChessBoard ();

    string getMove ();
    ChessPiece getPiece (int x, int y);
    void makeMove (int from_x, int from_y, int to_x, int to_y);

private:
    PlayerColor _board[8][8];
    string _whose_move;
    int _move_count;
};

ChessBoard::ChessBoard ()
    // 跟在冒号后面的是变量的列表，带着传递给构造函数的参数
    : _whose_move( "white" )
    , _move_count( 0 )
{
}
```

25.1.3 用初始化列表初始化常量字段

如果定义了类中的一个字段为常量，那么这个字段就必须在初始化列表中完成初始化工作：

```
class ConstHolder
{
public:
    ConstHolder (int val);

private:
    const int _val;
};

ConstHolder::ConstHolder ()
    : _val( val )
{}
```

你无法通过直接赋值来初始化一个常量字段因为那些常量字段都已经被固化了。初始化列表是类尚未完全形成的唯一的地方，所以在这里设置一些不可改变的对象是安全的。同样道理，如果你有个字段是引用类型的，那么它同样必须在初始化列表中完成初始化的操作。

在讲到继承的时候我们会学到初始化列表的另一个用途。

25.2 解构对象

正如同需要构造函数来初始化一个对象一样，有时你也需要有代码来清理那些不再需要使用的对象。举个例子，如果构造函数申请分配了内存（或者其他的任何资源），然后当你的对象不再使用的时候，这些资源最终需要归还给操作系统。进行这种清除的操作称为摧毁对象，它是在一个叫做*析构方法*的特殊的方法内部发生的。在一个对象不再需要的时候会调用析构方法——例如在对指向一个对象的指针调用 `delete` 时。

我们来看一个例子，假设有个类用来表示一个链表。要实现这个类，可能需要有一个字段来存储列表当前的头节点：

```
struct LinkedListNode
{
    int val;
    LinkedListNode *p_next;
};

class LinkedList
{
public:
    LinkedList (); // 构造函数
    void insert (int val); // 插入一个节点

private:
    LinkedListNode *_p_head;
};
```

如之前所见到的，链表中的头节点就如同别的元素一样，指向使用new关键字来分配出的内

存。这表示在某个时候，如果不再需要使用这个LinkedList对象了，我们要有一个清理它们的方式。这就是析构函数要干的活。来看看为这个数据类型加一个析构函数会是什么样子。和构造函数一样，析构方法也有个特殊的名称：就是在类的名字之前加一个波浪号（~），如同构造函数，析构函数也没有返回值。和构造函数所不同的是，析构函数永远不会接收任何参数。

```
class LinkedList
{
public:
    LinkedList (); // 构造函数
    ~LinkedList (); // 析构函数，注意波浪号(~)

    void insert (int val); // 插入一个节点

private:
    LinkedListNode *_p_head;
};

LinkedList::~LinkedList ()
{
    LinkedListNode *p_itr = _p_head;
    while ( p_itr != NULL )
    {
        LinkedListNode *p_tmp = p_itr->p_next;
        delete p_itr;
        p_itr = p_tmp;
    }
}
```


析构函数的代码和之前见过的删除链表中所有条目的代码相似，唯一不同的就是利用了一个类中的一个特殊方法来专门做清理工作。但是等等， 每个节点都去清除它自己的数据不是更有意义吗？这难道不是析构函数存在的所有意义吗？如果我们这么做会怎样呢？

```
class LinkedListNode
{
public:
    ~LinkedListNode ();
    int val;
    LinkedListNode *p_next;
};

LinkedListNode::~LinkedListNode ()
{
    delete p_next;
}
```

不管你信不信，这段代码触发了一系列的函数递归调用。这里发生的事是，使用delete就调用了p_next所指向的对象的析构函数（或者如果p_next为空的话就什么都不做）。那个被调用的析构函数紧接着又去调用delete也就是调用下一个析构函数。但是基本案例是怎样的呢？这一系列的解构器调用如何结束呢？最终p_next将会为空，在那个时候调用delete就什么也不做了。

所以是有个基本的案例存在的——它只不过正好被隐藏在对delete的调用之中了。一旦我们的LinkedListNode有了这个解构器，LinkedList自己的解构器只需要简单地加上这句代码：

```
LinkedList::~LinkedList ()
{
    delete _p_head;
}
```

这里调用delete开始了递归链，直到链表的最后。

现在你可能在思考——这么做是个很好的模式，但是为什么需要一个解构器呢？难道我们就不能写个自己的方法然后按喜好来命名它吗？当然可以，但是使用解构器有个好处：在对象不再需要的时候它会被自动调用。

那么说一个对象“不再需要了”到底是什么意思呢？它意味着下面三种情况中的一种：

(1) 当你删除了一个指向对象的指针；
(2) 当这个对象超出了作用域；
(3) 当拥有这个对象的类的析构函数被调用了的时候。

25.2.1 delete时的解构

调用delete很明显地反应了什么时候会调用析构函数，就如同你已经见过的：

```
LinkedList *p_list = new LinkedList;
delete p_list; // p_list的~LinkedList(析构函数)被调用了
```

25.2.2 超出作用域时的解构

第二种情况，一个对象超出了作用域，这是个隐含的操作。每当对象声明在大括号中时，在括号结束以后它们就超出作用域了。

```
if ( 1 )
{
    LinkedList list;
} // 链表的析构函数在这里调用
```

有种稍微复杂一点的例子就是当一个对象是在函数内部声明的时候。如果函数有返回语句，析构函数就会作为离开函数所进行的操作的一部分来调用。我想，对于在代码块中声明的对象的析构函数，它是在程序离开该代码块时“在走到右括号的地方”执行的。代码块的结束是在最后一个语句执行完毕的时候，或者由一个return语句或者break语句来实现退出代码块：

```
void foo ()
{
```

```
    LinkedList list;

    // 一些代码……
    if ( /* 某个条件 */ )
    {
        return;
    }
} // 链表的析构函数在这里调用
```

这种情况下，即使return是在if语句当中的，我也认为析构函数在函数走到最后一个大括号时才运行。但是，对你而言要掌握的最重要的是析构函数只在对象超出作用域时才执行——当它一被引用就出现编译错误的时候。

如果在某段代码块的末尾有多个对象需要执行解构器的话，那些解构器的运行顺序是正好与对象们的构建顺序相反的。举个例子，在下面的代码中：

```
{
    LinkedList a;
    LinkedList b;
}
```

b的解构器是在a的解构器之前执行的。

25.2.3 由其他析构函数导致的解构

最后，如果有个对象包含在另一个类当中，那个对象的析构函数是在类的析构函数调用之后被调用的。举个例子，如果你有个很简单的类：

```
class NameAndEmail
{
/* 正常情况下这里会有一些方法 */
private:
    string _name;
    string _email;
};
```

在这里，_name和_email字段的析构函数会在NameAndEmail的析构函数运行结束时被调用。这很方便——你无需做任何特殊的操作来清理类中的任何对象！你真的只需要调用一下delete来清理那些指针（或者别的资源如文件引用或者网络连接）。

顺便说一下，即使没有给类加个析构函数，这种情况下编译器同样会确保去执行你类中所有对象的析构函数。

使用构造函数来初始化一个类并且使用析构函数来清理属于这个类的内存或者别的资源，这个思想有个名称：*资源分配既初始化*或者叫**RAII**。基本的意思就是在C++中，你应该通过创建类来处理资源，并且在你创建类的时候，构造函数应当负责所有初始化的工作同时析构函数需要处

理所有的清理工作。不应该要求使用这个类的人去做什么特定的处理。通常，这会导致像上面NameAndEmail那样的类：两个字符串在完成使命以后会自己进行清理，这样NameAndEmail自身就不需要来实现析构函数了。

25.3　复制类

我们关于类的重要概念之旅的第三站就是处理复制类的实例。在C++中，创建可供复制的新类是经常要做的事——举个例子，你可能会这样写：

```
LinkedList list_one;
LinkedList list_two;

list_two = list_one;
LinkedList list_three = list_two;
```

在C++中，有两个函数可以定义用来确保这些复制操作能正常运行。一个函数是赋值操作符，另外一个是复制构造函数。我们先看一下赋值操作符，然后再讨论复制构造函数。

你可能会疑惑：为什么需要这些函数，不是直接写就可以了吗？答案是可以直接写，有时候就管用，因为C++ 会提供默认版本的复制构造函数和赋值操作符。

然而，有些情况下不能依赖默认的版本——有时编译器也不是那么聪明，它可能不知道你的意图。例如，默认版本的复制构造函数和赋值操作符会执行叫做*浅层指针复制*的操作。浅层指针复制就是将第二个指针赋值让其指向第一个指针所指向的内存地址。这种操作称为浅层是因为那些被指向的内存并没有被复制，复制的仅仅是指针而已。有时浅层复制可能是没问题的，但是有些情况下它就会导致问题。

举个例子，用之前的LinkedList类写下面的这些代码：

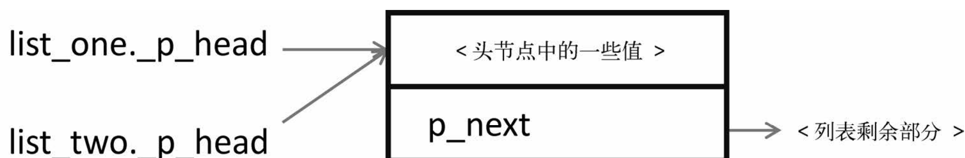
```
LinkedList list_one;
LinkedList list_two;

list_one = list_two;
```

这里的问题在于默认的赋值操作符会生成下面这样的代码：

```
list_one._p_head = list_two._p_head;
```

你可以像下面这样用图来描述这个过程：



现在两个对象有着相同的指针值，而且每个对象的析构函数都会试图释放同一个指针所指向的内存。

当list_two的析构函数运行的时候，它会删除list_two._p_head。（list_two的析构函数会先运行因为析构函数的运行顺序和构造函数正好相反，这里list_two的构造函数是第二个运行的。）然后list_one的析构函数会接着运行，去删除list_one._p_head。问题出现了，list_two._p_head已经被删除了，而如果要删除同一个指针两次，你的程序就要崩溃了！

很明显一旦其中的一个析构函数运行过以后，另外一个链表就不可用了！赋值操作符正好是处理这种问题的一个方式。所以，我们来看看它到底是什么样子的。

25.3.1 赋值操作符

在将一个对象赋值给一个已经存在的对象时赋值操作符会被调用，比如这么写的时候：

```
list_two = list_one;
```

要实现赋值操作符，需要少量的可以用来定义操作符的新语法。所幸，这还不是太麻烦：

```
LinkedList& operator= (LinkedList& lhs, const LinkedList& rhs);
```

这跟普通的函数声明看上去很像——它接收两个参数：一个是LinkedList的非常量引用另一个是LinkedList常量引用，并且返回一个LinkedList的引用。唯一怪异的地方就是函数的名字：operator=。这里的意思不是定义一个新函数，我们是定义了等号在LinkedList类中的用法。第一个参数是在等号左边的，也就是被赋值的，所以它不是常量。第二个参数是等号右边的，它是要赋给左边的值（并且它应当是常量，因为你没有理由要去修改它，尽管并没有严格要求它是常量）：

```
lhs = rhs;
```

之所以返回一个LinkedList引用，因为这样你可以将赋值语句链接起来：

```
linked_list = lhs = rhs;
```

现在，大多数时候，一个类会特地将operator=函数作为它的成员函数而不是一个单独存在的函数，这样operator=就可以操作类的私有字段了（相对于我上面那样只是声明一个游离于类之外的函数）。来看看具体的代码：

```
class LinkedList
{
public:
    LinkedList (); // 构造函数
    ~LinkedList (); // 析构函数，注意波浪线
    LinkedList& operator= (const LinkedList& other);

    void insert (int val); // 插入一个节点
```

```
private:
    LinkedListNode *_p_head;
};
```

注意少了一个参数：这是因为类的所有成员函数都隐式地将该类作为一个参数。在这里，operator=方法是在该类自身作为赋值操作左边元素的时候使用的。换句话说，在代码里这样表示：

```
lhs = rhs;
```

operator=函数是在变量lhs身上调用的。就如同这样写：

```
lhs.operator= ( rhs );
```

在函数执行完毕以后，lhs就会和rhs有相同的值。好，那么我们就来谈谈如何为LinkedList类写个operator=函数。

```
LinkedList& LinkedList::operator= (const LinkedList& other)
{
    // 这里会是什么呢
}
```

通过上面的讨论，我们已经知道仅仅复制指针地址并不完全正确。

我们真正要做的是复制整个结构。逻辑是这样的：首先释放已有的列表（因为它已经不需要了），然后复制每个列表节点，这样就有了两个相互独立的列表。最后，由于需要返回一个值，我们会返回被复制的这个类的一个副本。

最后一步需要一个新的语法——需要有一些指向当前对象的方式。在C++中要实现这个功能，我们可以使用一个特殊的变量，叫做this指针。this指针是指向当前类的实例的一个指针。例如，假使写list_one.insertElement(2)；那么在insertElement内部，你就可以使用关键字this，它指向list_one。我们还将使用this指针为方法增加一点安全性。

```
LinkedList& LinkedList::operator= (const LinkedList& other)
{
    // 确保不是将自己赋值给自己，如果出现这种情况就忽略它
    // 注意，这里使用 'this' 来确保另外一个值和我们的对象不是同一个地址
    if ( this == & other )
    {
        // 返回this对象来维持赋值链不被破坏
        return *this;
    }
    // 在复制过来新的值时，我们需要释放原来的内存，因为它没用了
    delete _p_head;
    _p_head = NULL;


    LinkedListNode *p_itr = other._p_head;
```

```
        while ( p_itr != NULL )
        {
            insert( p_itr->val );
        }
        return *this;
    }
```

这个函数有几个注意点：首先，注意我们做了自身赋值的检查——自身赋值是那种通常情况下你不希望碰到的，但是没有理由不去确保这个操作是安全的。像下面这样写：

```
a = a;
```

应该是完全没问题的，并且不改变任何东西。

接下来，我们需要释放原来的列表所占用的内存，因为已经不用它了：删除_p_head就可以删除整个列表，就像在析构函数里一样。

最后，我们要使用右边的新数值来重新生成列表，可以通过循环遍历整个旧的列表然后把它的每一个值都插入到自己的列表中。现在看看，我们有一个可以复制的类了！

所幸，不是所有的类都需要这样复杂的复制操作。如果类的成员中没有一个是指针，你可能根本就不需要一个赋值操作符！没错——这就是C++，仁慈而细心，它会默认提供一个赋值操作符，该赋值操作符会通过运行每个元素自身的赋值操作符（如果这个元素是一个类的对象）或者复制它的值（如果这个元素是指针或者别的数值）。所以如果类中没有指针，在大多数情况下你都可以依赖默认的赋值操作符。有个好的法则，那就是如果需要写自己的析构函数，那么你恐怕也要自己写赋值操作符。这个法则的道理是如果你有自己的析构函数，那么它可能是用来清理释放内存的，而如果有释放内存的操作，就需要确保类的副本都有它们自己的内存。

25.3.2 复制构造函数

最后还有一种要考虑的情况，假使你想要依照另一个对象来构造一个相同的对象会怎样：

```
LinkedList list_one;
LinkedList list_two( list_one );
```

这只是构造函数使用的一个特殊情况——构造器接收的参数是和正在构造的对象属于同一类型的对象。这样的构造函数称为*复制构造函数*。复制构造函数应当能够使新的对象是原有对象的一个直接复制。这里就是list_two应当初始化成和list_one一模一样。这有点像赋值操作符，除了这里是直接从一个未初始化的类开始的操作而不是已经有一个类存在了。这是个好事因为它意味着无需浪费任何CPU资源来构建类，你只要重写一下相应的值就行了。复制构造函数通常实现起来很简单并且看上去和赋值操作符很像。对于LinkedList它是这样的：

```
class LinkedList
{
```

```
public:
    LinkedList (); // 构造函数
    ~LinkedList (); // 析构函数，注意波浪号
    LinkedList& operator= (const LinkedList& other);
    LinkedList (const LinkedList& other);


    void insert (int val); // 插入一个节点

private:
    LinkedListNode *_p_head;
};

LinkedList::LinkedList (const LinkedList& other)
    : _p_head( NULL ) // 默认是NULL，以防另一个列表是空的
{
    // 注意，这段代码和operator=很像
    // 在正式的程序中写个辅助性的方法来做这件事是有意义的
    LinkedListNode *p_itr = other._p_head;
    while ( p_itr != NULL )
    {
        insert( p_itr->val );
    }
}
```

看到了没？小菜一碟儿。

如果你自己不写的话，编译器会提供一个默认的复制构造函数。这个默认的复制构造函数所做的操作和默认赋值操作符一样：它会执行类的每个对象各自的复制构造函数，并且它对像整型和指针这样的值会进行常规的复制。大多数情况下，如果需要自己实现一个赋值操作符，你恐怕也要顺带实现一个复制构造函数。

关于复制构造函数有件事你需要知道，它有时会惊呆初学的小伙伴——当然在第一次遇到的时候它也惊到我了。

如果写了下面的代码：

```
LinkedList list_one;
LinkedList list_two = list_one;
```

你觉得会发生什么——它会调用赋值操作符吗？不，结果是编译器足够智能可以识别出list_two正在基于list_one进行初始化，实际上它会给你调用复制构造函数，免去一个没有必要的对象初始化。这是不是很好呢？

25.3.3　所有编译器生成的方法

现在你已经见过编译器自动生成的每一个方法了：

(1) 默认构造函数；

(2) 默认析构函数；
(3) 赋值操作符；
(4) 复制构造函数。

对于创建的每一个类，你都应当考虑一下是否能接收编译器默认为你实现的这些方法。很多时候可以用它们，但是如果你有指针需要操作，就经常要声明自己的析构函数，赋值操作符以及复制构造函数。（通常情况下，如果需要它们中的某一个，那么你就需要它们所有的。）

25.3.4 彻底地阻止复制

有些时候根本不需要复制对象的功能。“不许这个对象被复制”，这么说不也很好吗？这么做可以避免实现复制构造函数或者赋值操作符，并且也不要承担编译器会生成这些方法的危险版本的风险。

也有些情况下复制对象就是错误的。举个例子，假设有个游戏其中一个类代表当前玩家的飞船，你实在不想这个飞船有另外的副本——只想要个唯一的飞船，且其中包含了当前玩家的所有信息。

你可以通过声明复制构造函数和赋值操作符，却不去实现它们来做到阻止复制。一旦你声明了一个方法，编译器就不再自动生成它了。如果试图去使用它，将会在链接时得到一个错误因为你使用了一个未定义的函数。这可能有点让人费解，因为链接器不会告诉你问题到底出在哪一行代码上。你也可以通过把这些方法设为私有来获得更好的报错信息；这样，大部分情况下错误就会发生在编译的阶段，可以给出更容易理解的错误信息。来看看具体怎么做：


```
class Player
{
public:
    Player ();
    ~Player ();

private:
    // 通过声明却不定义这些方法来，然后编译器不会为我们自动生成，
    // 这样就禁止了复制操作
    operator= (const Player& other);
    Player (const Player& other);

    PlayerInformation *_p_player_info;
};

// 没有赋值操作符或者复制构造函数相关的实现
```

总结一下，应当总是选择下面这些操作中的一个：

(1) 同时使用默认的复制构造函数和赋值操作符；
(2) 同时创建自己的复制构造函数和赋值操作符；
(3) 将复制构造函数和赋值操作符都设为私有，并且不去实现它们。

如果你什么都不做，由于编译器的默认生成你相当于选择了第一个选项。通常最简单的是选择第三种方案，然后如果发现有需要时再去实现赋值操作符和复制构造函数。

25.4　问答题

(1) 你在什么时候需要给类写一个构造函数？

A. 总是需要写，没有构造函数你就不能使用这个类
B. 在你需要以非默认值来初始化类的时候
C. 永远不需要，编译器总是会为你提供一个
D. 只有你同时需要一个析构函数的时候

(2) 析构函数和赋值操作符之间的关系是什么？

A. 它们没什么关系
B. 类的析构函数会在运行赋值操作符之前被调用
C. 赋值操作符需要指出哪些内存应当被析构函数删除掉
D. 赋值操作符必须确保运行被复制类的析构函数和运行新类的析构函数都是安全的

(3) 在什么时候需要使用一个初始化列表？

A. 在你想要让构造函数尽可能地高效以及想要避免构造空的对象时
B. 在你初始化一个常量时
C. 在你想要运行类的某个字段的非默认构造函数的时候
D. 上面所有的都成立

(4) 下面代码的第二行执行时哪个函数会运行？

```
string str1;
string str2 = str1;
```

A. `str2`的构造函数和`str1`的赋值操作符
B. `str2`的构造函数和`str2`的赋值操作符
C. `str2`的复制构造函数
D. `str2`的赋值操作符

(5) 下面代码中哪些函数被调用了，顺序是怎样的？

```
{
        string str1;
        string str2;
}
```

A. str1的构造函数，str2的构造函数
B. str1的析构函数，str2的构造函数
C. str1的构造函数，str2的构造函数，str1的析构函数，str2的析构函数
D. str1的构造函数，str2的构造函数，str2的析构函数，str1的析构函数

(6) 如果已知一个类有个非默认的构造函数，下列关于它的赋值操作符哪个应该是正确的？

A. 它应当有个默认的赋值操作符
B. 它应当有个非默认的赋值操作符
C. 它应当有个声明了但是没有实现的赋值操作符
D. B 或 C 正确

25.5 实践题

(1) 实现一个 vector 的替代品，让其只能操作整型，叫vectorOfInt（需要像正式的STL那样使用模板）。类应当包含下列这些接口：

- 一个分配32个元素vector的无参数构造函数
- 一个接收初始化大小作为参数的构造函数
- 一个get方法，接收一个索引并返回该索引对应的值
- 一个set方法，接收一个索引和一个值，将值设为索引对应的值
- 一个pushback方法，向数组的末尾添加一个元素，如有必要重新给数组定义大小
- 一个pushfront方法，向数组的开头添加一个元素
- 一个复制构造函数以及一个赋值操作符

类应该不存在内存泄漏；任何分配的内存都应当被释放。试着仔细想想类可能被怎样误用，以及你该如何处理这些误用。如果用户给了个负的初始化大小你要怎么做？如果用户访问了负的索引值怎么办？

第 26 章 继承和多态

26

到目前为止我们一直在讨论如何通过能提供干净利落的公共接口和具有对像新建、复制、清除功能的类，来创造一个完整的，有用的数据类型。现在让我们来更进一步地探讨一下接口的思想。假设你有一辆汽车，它有点破旧缓慢。遗憾的是，几乎每家汽车厂商都有各自不同的控制机制——有些厂商使用方向盘，有些使用操纵杆，有些使用鼠标，有些有油门，有些则需要你拖拽滑动条[①]。这不是很可怕吗？每次要使用汽车，你都得重新去学习怎么控制它。每次想要租或者买一辆新车，你都得重新学如何驾驶它。

所幸的是，汽车都会追随一定的标准。任何时候你上了一辆车，它都是同样的接口——方向盘，油门。唯一的不同就是有些车是自动挡，有些车是手动档。一辆车可以有两个接口：自动或者手动。

只要知道怎么使用自动挡，你就可以驾驶任何自动挡的车。在你开车的时候，引擎的细节并不重要。重要的是它要提供和其他汽车相同的打方向、加速以及刹车的方法。

这些和C++有什么关系呢？在C++当中，事实上写代码的时候希望有特定的，定义良好的接口（用上面的来作类比，你就是代码，汽车的驾驶机制就是接口）来直接使用是可能的。接口（汽车自身）的实现并不重要——接口的任何特定的实现（任何你选择的汽车），都可以为外部代码（被你，司机）所用，因为它实现了一个代码能够理解的接口。你，作为司机，可能相比于一些车来说更喜欢另外一些车，但是你都可以驾驶它们。

好了，在什么时候会写具有和上面相同性质的代码呢？假设有一个游戏——你可能有很多不同的对象需要绘制到屏幕上——子弹、飞船、敌人。在游戏的主循环中，每一帧你都要把它们中的每个绘制到各自新的位置上。

真想能写出下面这样格式的代码：

```
清除屏幕
遍历可以绘制的对象列表
    对于每个可绘制对象，绘制它
```

① 当然，用滚轮来操作汽车可能会造成很多的事故。

可绘制对象列表理想状态下可以存储各种你可以绘制到屏幕上的对象。它们都需要实现一些通用接口，这些接口可以允许把它们绘制到屏幕上。但是你还想让子弹、飞船和敌人各自是一个不同的类——它们有各自不同的内部数据（玩家的飞船需要有生命值，敌人的飞船需要有AI来移动它们，而子弹则需要存储它所能造成的伤害）。

对于绘制对象的循环来说，这些具体的东西都是无关紧要的。最重要的是这些不同的类都要支持一个允许绘制的接口。

怎样来做到这些呢？首先，来定义一下怎么才叫做可以被绘制：

```
class Drawable
{
public:
    void draw ();
};
```

这个简单的Drawable类，只定义了单独的一个方法——draw。这个方法绘制当前的对象。如果写个vector<Drawable*>，然后把所有实现了draw方法的东西都存放在其中，这种做法是不是很好呢[①]？如果可以这么做，我们就可以写代码通过遍历vector中所有的东西，调用draw方法来将它们都绘制到屏幕上。

任何使用vector中存储的对象的人都只能使用那些构成Drawable接口的方法，但是不管怎么说，这就是这里所要做的一切。

你猜怎么着？C++事实上就支持这个！让我们来看看如何实现它。

26.1　C++中的继承

首先，我们介绍一个新的术语：*继承*。继承的意思是一个类从另一个类那里获得一些特性。在上面的例子里，被继承的特性将会是Drawable类的接口，具体地说就是draw方法。一个从别的类那里继承特性的类称为*子类*。被继承的那个类是*父类*[②]。一个父类通常会定义一个接口方法（或者多个方法），这些方法可以被各个子类以不同的方式来实现。在我们的例子中，Drawable就是个父类。游戏中每个Drawable对象都将是Drawable的子类；每个类都会继承拥有draw方法这一特性，让获取Drawable对象的代码能够知道draw方法是可用的。然后每个类都会实现它们自己的draw方法版本——实际上，它*必须*实现自己的draw方法版本，要保证Drawable的所有子类都有一个正确的draw方法。

好了，了解基本概念了吧？继续来看看语法：

---

① 如果你在纳闷我何以把指针放入vector中，这里的原因就是我们需要使用指针来获取我们将要看到的操作。

② 有时使用*超类*来代替父类，使用派生类代替子类。本书中将使用父类和子类。使用父类和子类可能比较符合习惯。

```
class Ship : public Drawable
{
};
```

: public Drawable表示Ship类继承自Drawable类。这么写，Ship从它的父类也就是Drawable那里继承了所有的公共方法和公共数据。现在，Ship就已经继承了draw方法。实际上是整个方法，包括实现。如果这样写：

```
Ship s;
s.draw();
```

对draw方法的调用会调用到写在Drawable里面的draw方法的实现。在这里这不是我们想要的，因为Ship类应当有它自己的绘制方式，而不是使用用来作为Drawable接口一部分的那个版本。

要让Ship类能够实现这个想法，Drawable类必须标示draw方法可以被子类重写。你可以使用*虚方法*（virtual），虚方法是父类的一个组成部分，但是它可以被不同的子类所重写。

```
class Drawable
{
public:
    virtual void draw ();
};
```

很多情况下，你并不需要父类提供任何的具体方法实现，而是需要强制子类要有它们自己的实现。（比如说，并不存在一个“默认”的方式来绘制一个对象。）你可以通过把函数设为纯虚函数来达到强制目的，就像下面这样（注意那个 = 0）：

```
class Drawable
{
public:
    virtual void draw () = 0;
};
```

这个语法第一次看上去肯定是怪异的！尽管这么写是遵循逻辑的——把方法设为0是表示它不存在的一种方式。如果一个类有纯虚方法，那么它的子类就必须实现这个纯虚方法。要实现它，子类需要再次声明这个方法，不要在后面加= 0。这表示该类将会提供一个这个方法的真正的实现：

```
class Ship : public Drawable
{
public:
    virtual draw ();
};
```

现在这个方法就可以像任何普通的方法一样来定义了：

```
Ship::draw ()
{
```

```
    /* 执行绘制的代码 */
}
```

你也许会问，如果所要做的只是让draw方法没有任何实现，那么到底为何还需要一个像Drawable这样的父类？关键点就在于需要父类是为了定义所有子类都要实现的接口。然后我们就能写代码，这些代码准备着使用Drawable接口而不需要知道正在使用的类到底是什么类型的。有些编程语言允许你把任何对象传递给任何函数，并且只要传进去的对象实现了该函数需要使用到的方法，一切都能正常运行。然而，C++要求函数公开它们参数的接口。如果我们没有Drawable接口，开始甚至都不能把这些类都放到同一个vector中；没有任何“共同的”东西可以用来识别哪些可以放进vector中。来看看使用vector并且绘制所有对象的代码：

```
vector<Drawable*> drawables;

// 通过创建一个新的 Ship 指针把 Ship 存储在 vector 中
drawables.push_back( new Ship() );

for ( vector<int>::iterator itr = drawables.begin(), end = drawables.end();
    itr != end; ++itr )
{
    // 当有一个指向对象的指针时，记住我们需要使用 -> 语法来调用方法
    (*itr)->draw(); // 调用 Ship::Draw
}
```

我们可以把不同类型的Drawable对象添加到 vector （假设有个同样继承自Drawable的Enemy类）：

```
drawables.push_back( new Ship() );
drawables.push_back( new Enemy() );
```

一切都会正常运行——对于飞船我们调用的是Ship::draw，而对于敌人调用的是Enemy::draw。

顺便说一下，我们使用vector<Drawable*>而不是vector<Drawable>这一点很重要。指针有着很大的意义；如果不是用指针，这些都将歇菜。

要看看为什么，比如我们写下不是使用指针来保存对象的代码时：

```
vector<Drawable> drawables;
```

在内存中，我们现在会开辟存储着不同Drawable对象的内存，所有的都是相同大小：

```
[Drawable 1][Drawable 2][Drawable 3]
```

如果不是使用指针的话 vector 就必须要存储下整个的对象。但是每个对象可能大小并不一样——一个Ship和一个Enemy可能有不同的字段，并且可能都比基本的Drawable小。这样代码就不能正确地运行了。

相反，指针一直是相同的大小[①]。我们可以这么说：

```
[Pointer to Drawable][Pointer to Drawable][Pointer to Drawable]
```

如果有一个[Pointer to Ship]，它所要占的内存和指向一个Drawable的指针是完全一样大的。这也是为什么要这么写：

```
vector<Drawable*> drawables;
```

现在我们可以凭意愿把任何类型的指针放到vector中，只要这个指针是指向一个继承自Drawable的类的，在循环之内，所有这些对象都会被绘制到屏幕上，使用的是子类的draw方法。（从技术上讲，任何指针都是合法的，但是不能仅仅因为它合法就把它放到我们的vector当中。这里vector的全部意义就在于存放一系列可以被绘制的东西。放进来一些无法被绘制的东西将会成为可怕的麻烦。）

要记住：任何时候想要用一个继承了父类接口的类，你都需要使用指针来传递它。

现在既然都已经看过了这个例子的所有细枝末节，那么回头来看看我们都做了些什么。

(1) 首先定义了一个Drawable接口，它可以被子类继承。

(2) 任何把Drawable当做参数的函数，或者任何可以使用Drawable的代码，都可以调用其所指向的对象实现的draw方法。

(3) 这允许已有的代码使用新类型的对象，只要这些对象实现了Drawable接口。我们可以向游戏中添加新的东西——增加力量的钱币或者额外的生命，背景图片，无论什么——处理它们的代码除了要求它们是Drawable之外不需要知道关于它们的任何情况。

这些都涉及重用。这里的重用指的是自已有的代码可以对新创建的类进行操作。可以写新的类，和已有代码（如绘制游戏中各个元素的循环）兼容，而不需要修改已有代码来配合新的类。（我们确实需要把新类的对象加到存放Drawable的vector中，但是循环本身不需要修改。）

这种行为叫做多态。顾名思义，多就是代表很多，而态呢就代表格式形态——合起来就是很多形态。换句话说，每个实现特定接口的类就是一种形态，并且由于有些代码写出来就是仅仅为了使用接口的，这样它们就能处理很多不同的类，这些代码也就可以支持特定接口的多种形态——就如同一个会开车的人就能开汽油动力的，混合动力的，或者是纯电动的汽车。

26.1.1 继承的别的作用以及误用的情况

多态取决于继承，但是继承并不是仅仅可以继承一个接口。如我之前提到的，还可以利用继承来获得一个函数实现。

① 对我们的目的来说这几乎就是对的了。有些机器可能不同的数据类型会有不同的指针，这里不要去担心它。如果你好奇，可以了解更多：http://stackoverflow.com/questions/1241205/are-all-data-pointers-of-the-same-size-in-one-platform。

举个例子，如果Drawable接口还有另外一个非虚方法，这个方法将会被每个继承Drawable的对象继承过去。有时候人们相信继承是为了通过继承方法来实现重用（这样避免了为每个子类都去写同样的方法）。然而这是一个局限性很大的重用方式。你确实可以通过继承完整的方法实现来节省一些时间或者空间；但是如果这么做了，那么你就有了个大的挑战：怎么去确保那个方法的实现对于每个子类来说都是正确的呢？这需要仔细的思考是否某个东西一直都是正确的。

来看看为什么这很困难。假设有Player和Ship两个对象，它们都实现了Drawable接口，同时这些类都有个getName的方法。你也许要决定把getName方法添加到Drawable类当中，这样这两个类就可以共享这个方法相同的实现。

```
class Drawable
{
public:
    string getName ();
    virtual void draw () = 0;
};
```

由于getName不是虚的，所有的子类都会继承这个方法的实现。如果你决定要加入一个新的类，一个想要绘制出来的，比如说Bullet，会发生什么事？每个子弹都需要有个名字吗？当然不是！让Bullet类拥有一个没用的getName方法看上去似乎没什么大不了的，而对一个类来说，这不是多了一个不好的方法这么简单。问题在于一次又一次地这么做会造成令人费解的复杂的类层级，这时候接口的目的就显得不太清楚了。

26.1.2 继承、对象构建和销毁

当继承一个父类的时候，子类的构造函数会调用父类的构造函数——就像它调用类的所有字段的那些构造函数一样。

举个例子，看看下面的代码：

```
#include <iostream>

using namespace std;

class Foo // Foo在计算机编程中是个常用的占位符
{
public:
    Foo () { cout << "Foo's constructor" << endl; }
};

class Bar : public Foo
{
public:
    Bar () { cout << "Bar's constructor" << endl; }
};
```

```
int main ()
{
    // 一个可爱的小东西 ;)
    Bar bar;
}
```

示例代码59：constructor.cpp

在bar被初始化的时候，首先Foo的构造函数会运行然后Bar的构造函数再运行。这段代码的输出是：

```
Foo's constructor
Bar's constructor
```

让父类的构造函数先运行，这样可以在子类可能使用父类的字段之前，先初始化父类的所有字段。在运行子类的构造函数之前运行父类的构造函数可以确保在子类可能使用父类字段的时候，事先知道那些字段都已经初始化过了。

这些工作编译器都自动为你做好了——你不需要做任何事来让父类的构造函数被调用。相似地，在子类的析构函数运行以后，父类的析构函数会被自动调用。下面这段代码就是一个例子：

```
#include <iostream>

using namespace std;

class Foo // Foo在计算机编程中是个常用的占位符
{
public:
    Foo () { cout << "Foo's constructor" << endl; }
    ~Foo () { cout << "Foo's destructor" << endl; }
};

class Bar : public Foo
{
public:
    Bar () { cout << "Bar's constructor" << endl; }
    ~Bar () { cout << "Bar's destructor" << endl; }
};

int main ()
{
    // 一个可爱的小东西;
    Bar bar;
}
```

示例代码60：destructor.cpp

这里是上面代码的输出：

```
Foo's constructor
Bar's constructor
Bar's destructor
Foo's destructor
```

注意，构造函数和析构函数被调用的顺序是相反的；这样可以保证Bar's的析构函数能够安全地使用继承自Foo的方法，因为那些方法操作的数据仍然处在一个合法、可用的状态。这和父类构造函数要在子类构造函数之前运行背后的理由是很相似的。

在有些情况下，你也许希望调用一个非默认的父类构造函数。初始化列表可以允许你这么做，通过在列表中提供父类的名字来实现。

```
class FooSuperclass
{
public:
    FooSuperclass (const string& val);
};
class Foo : public FooSuperclass
{
public:
    Foo ()
        : FooSuperclass( "arg" ) // 初始化列表示例
    {}
};
```

父类构造函数的调用在初始化列表中应当出现在该类的字段之前。

26.1.3 多态和对象销毁

对象销毁以及当一个对象通过接口被销毁的时候它是如何运行的，这些是容易弄错的地方。举个例子，你可能写了像下面这样的代码：

```
class Drawable
{
public:
    virtual void draw () = 0;
};

class  MyDrawable : public Drawable
{
public:
    virtual void draw ();
    MyDrawable ();
    ~MyDrawable ();

private:
    int *_my_data;
};
```

```
MyDrawable::MyDrawable ()
{
    _my_data = new int;
}

MyDrawable::~MyDrawable ()
{
    delete _my_data;
}

void deleteDrawable (Drawable *drawable)
{
    delete drawable;
}

int main ()
{
    deleteDrawable( new MyDrawable() );
}
```

那么在deleteDrawable里面会发生什么呢？记住析构函数是在delete使用的时候被调用的。

那么这行代码：

```
delete drawable;
```

就是在调用该对象的一个函数。但是编译器怎么知道如何去找到MyDrawabe的析构函数呢？它并不知道这个drawable变量的具体类型——它只能知道这个变量是个Drawable一个含有名称为draw的方法的东西。它只知道如何去找到与Drawable相关联的析构函数，而不是MyDrawable自己的析构函数。不幸的是，这里由于MyDrawable类在它的构造函数中分配了内存，运行MyDrawable的析构函数来释放那段内存是很重要的。

你也许会想：这不正好是虚函数应该去处理的问题吗？没错，正是如此！我们所需要做的就是在Drawable类中把析构函数声明为虚的，这样在一个指向Drawable的指针被调用delete的时候，编译器就知道去寻找一个重写的析构函数了。

```
class Drawable
{
public:
    virtual void draw ();
    virtual ~Drawable ();
};

class MyDrawable : public Drawable
{
    public:
    virtual void draw ();
    MyDrawable ();
    virtual ~MyDrawable ();
```

```
private:
    int *_my_data;
};
```

通过在父类中将析构函数标志为虚的，在使用delete释放一个Drawable接口的时候，重写的析构函数就会被调用。

和通常的规则一样，当你把父类中的任何方法设为虚的时，就应该把父类的析构函数设为虚的。当你将一个单独的方法设为虚的时候，就是在说人们可以把这个类传到接收一个接口为参数的方法中。那些方法可以做任何它们想要做的，包括删除传进来的对象，所以将析构函数设为虚的用来保证对象会被正确地清理掉。

26.1.4 对象切割的问题

对象切割是在处理继承时需要注意的另一个问题。对象切割常在你写出下面这样的代码时发生：

```
class Superclass
{};

class Subclass : public Superclass
{
    int val;
};
int main()
{
    Subclass sub;
    Superclass super = sub;
}
```

26

来自子类的val字段并没有作为赋值操作的一部分赋给父类！遗憾的是，这通常不是你想要的（尽管事实上C++允许这种操作）因为在父类变量那里这个对象只有一部分。这种类型的切割有时能够运行，但是它会经常导致程序崩溃[①]。

幸运的是，有个方式可以让编译器告诉你有这类问题发生。你可以把父类的复制构造函数声明为私有的并且不要去实现它：

```
class Superclass
{
public:
    // 注意，由于我们声明了复制构造函数，
    // 因此需要提供自己的默认构造函数
    Superclass () {}
private:
```

① 尤其是该类有想要使用子类字段的虚函数的时候。

```
    // 我们不会定义这个方法，这是被禁止的操作，
    Superclass (const Superclass& other);
};

class Subclass : public Superclass
{
    int val;
};

int main ()
{
    Subclass sub;
    Superclass super = sub; // 现在这行代码就会导致一个编译错误
}
```

但是假如真的需要有个复制构造函数怎么办呢？另外一种避免这个问题的方式是让所有的父类都至少有一个虚函数。这样可以保证就算你这么写：

```
Superclass super;
```

代码都不会通过编译，因为你不能创建一个有着纯虚函数的对象。另一方面，这么写仍然是可以的：

```
Superclass *super = & sub;
```

这样就可以利用多态的好处同时避免对象切割的问题。

26.1.5　与子类共享代码

到目前为止我们已经讨论了public修饰符和private修饰符的保护作——public方法对于类以外的任何人都是可用的，private方法和数据只对于同一个类当中的其他方法可用。

但是如果想要一个父类能够提供子类可以调用的方法，而又不是通过内部的类来实现，该怎么做呢？首先，你会有这么做的需求吗？可能会的。父类共享出一些实现的代码是很常见的事。

举个例子，假设有个通过清除屏幕上某块区域来帮助对象绘制自身的方法。我们称这个方法为clearRegion：

```
class Drawable
{
public:
    virtual void draw ();
    virtual ~Drawable ();
    void clearRegion (int x1, int y1, int x2, int y2);
};
```

这里继承的使用不是为了继承接口，而是为了子类能够访问通用的实现代码。这是继承的一个合法使用，因为子类要么需要使用这个方法要么可能需要使用它。由于它不是公共接口的一部

分，它只是所创建的类层级中的一个实现细节。

但是怎么来避免这个方法成为类的接口的一部分呢？如上面展示的一样，把它设为public允许任何人调用这个方法——哪怕它本不应该是这样的。另一方面，你又不能把它设为private，因为子类不能访问父类的私有字段和方法，而且阻止子类的访问会使我们的整个目的失败！

26.1.6　**protected**的数据

答案就是使用第三种也是最后一种访问修饰符——protected。任何在类的protected区域的方法都可以被子类访问，不像private方法那样，但是在类之外又是不可访问的，不像public方法那样。protected所用的语法和public与private是一样的：

```
class Drawable
{
public:
    virtual void draw ();
    virtual ~Drawable ();
protected:
    void clearRegion (int x1, int y1, int x2, int y2);
};
```

现在只有Drawable的子类可以访问clearRegion。

protected方法通常很有用，但是我可从来不会推荐使用protected修饰的数据。没有必要把数据的全部访问权限暴露给整个的类层级，原因和不想要把数据暴露到其他别的地方一样——因为想要在以后能够修改它。取而代之，可以使用protected方法来提供子类中对父类数据的访问。

26.1.7　属于类的数据

到现在为止，对一个类你所能做的都是把数据存储在单独的对象实例中。很多情况下，这就足够了，但是还有一些情况确实需要存储不仅仅是属于某个特定对象的数据，而是属于整个类的数据。有个例子就是如果想要创建一个类，它要求每个对象有个唯一的序列号。每个对象都应该有它自己的序列号，但是怎样跟踪想要赋值的下一个序列号呢？你需要有地方在类的层次上来存储“下一个序列号”，这样每当一个新的对象创建的时候，就知道要赋给它什么值。（为什么要做类似这样的事呢：首先有一点，使用每个对象的序列号能够更简单地在日志语句中识别这些对象。序列号可以用来在不同行数的日志文件中跟踪某个对象。）

你创建属于类的数据的方式是使用一个该类的静态成员。不像普通的实例数据，静态数据不是任何单个对象的一部分；它对于类的所有对象都可用，如果是public那就对所有人都可用。实际上，静态变量和全局变量很相似，除了在类的外部访问静态变量时你需要在变量名之前添加类名作为前缀。

来看看它写出来是什么样子。下面是声明了一个静态变量的类：

```
class Node
{
public:
    static int serial_number;
};
// 不是在类声明里，所以需要使用 Node:: 作为前缀
static int Node::serial_number = 0;
```

不仅可以使用静态变量，你还可以使用静态方法——作为类的一部分的方法，它可以在没有实例对象的情况下使用。让我们来看看通过添加一个叫做_getNextSerialNumber的私有静态方法来创建序列号。

```
class Node
{
public:
    Node ();

    private:
    static int _getNextSerialNumber ();

    // 静态的，整个类只有一份
    static int _next_serial_number;

    // 非静态的，对于每个对象都可用，但是不能被静态方法使用
    int _serial_number;
};

// 不是在类声明里，所以需要使用 Node:: 作为前缀
static int Node::serial_number = 0;

Node::Node ()
    : _serial_number( _getNextSerialNumber() )
{}

int Node::_getNextSerialNumber ()
{
    // 使用 ++ 在后面的方式来返回变量中的前一个值
    return _next_serial_number++;
}
```

要记住，当你使用静态方法的时候，它是类的一部分，但是它没有权限访问对象特有的字段。它只能访问静态的数据。静态方法在调用的时候没有this指针传递给它。

26.1.8　如何实现多态

注意：编译器怎么实现多态是个很高级的话题，同时它会带你深入到 C++ 在这方面的实现。我把这一部分包含进来是因为它是个优雅的实现技巧，而我忍不住不和你分享它。在你第一次（或者第二次）接触多态的时候就去学习这个知识并不是必要的。如

果你对于多态的魔法是怎么实现的感到好奇，那就继续读下去；如果头皮发麻，就不要再费劲了。以后你需要理解更多细节的时候，随时可以返回到这部分。

多态的核心思想是在接口上执行函数，而不是在一个具体的子类上，这样对应一行给出的机器码就不需要确切知道要调用哪个函数。举个例子，下面的代码中：

```
vector<Drawable*> drawables;

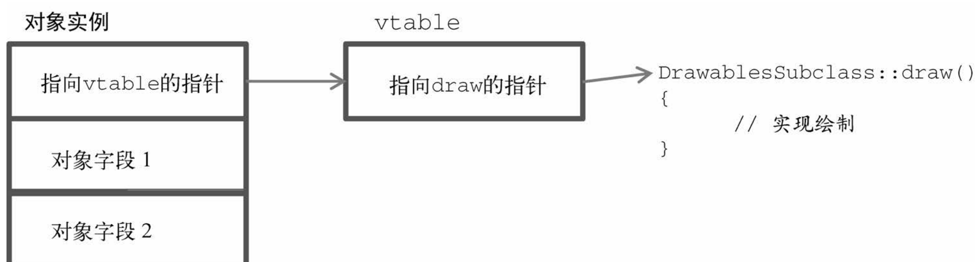
void drawEverything ()
{
    for ( int i = 0; i < drawables.size(); i++ )
    {
        drawables[ i ]->draw();
    }
}
```

对drawables[i]->draw()的调用无法被编译成对一个特定方法的调用，因为draw方法是虚的。根据不同的继承自Drawable的对象，它可能调用任意个不同的方法：绘制一发子弹，玩家的飞船，一个敌方飞船，或者一个大力药丸。

此外，drawEverything对于它调用的代码一无所知。调用draw方法的那行代码只要看到Drawable接口就行了。它不需要了解任何真正实现Drawable的对象。但是这样它又怎么去调用Drawable子类的方法呢？

对象持有一个虚方法的列表作为它的隐藏字段——在这个例子当中，有一个入口，含有draw方法的地址。接口中的每个方法都被赋予了一个数字（draw是方法 0）；当调用一个虚方法的时候，与该方法相关联的数字会被用来作为访问该对象虚方法列表的索引。对虚方法的调用被编译为一个对虚方法列表的查找，后面紧跟着一个对所查找的方法的调用。在上面的代码中，对于draw方法的调用会变成在方法表中对方法 0 的一个查找，跟随着对方法 0 的地址的调用。这里的虚方法列表称为vtable（虚拟表的简称）。

这里是它的示意图：



由于对象持有着它自己使用的方法表，编译器在编译不同的类的时候可以改变表中的地址来

提供虚方法的一个指定的实现。当然，这不用自己做——编译器都帮你做了。使用方法表的代码只需要确切地知道表中寻找每个虚方法的索引。

虚方法表只会包含那些被声明为虚的方法——非虚方法不需要这个机制，所以它们也就没有虚方法表的入口。如果你写的类根本就没有虚方法，那么它也就不会有虚表。

当一个虚方法被调用的时候，这就相当于执行访问虚表并且通过索引找到方法的代码。这样写：

```
drawables[ i ]->draw();
```

编译器会做如下理解。

(1) 获取存储在drawables[i]中的指针。
(2) 通过这个指针找到与Drawable类型的接口相关的那组方法所在虚表的地址（这个例子中只有一个方法）。
(3) 在函数表中找到给定名称（这里就是draw）的函数。函数表在字面上就是存储着每个函数在内存中的地址的集合。
(4) 带着相关的参数去调用所找到的函数。

通常第(2)步不是通过使用函数真正的名称来完成的，而是通过编译器把每个函数名转换为表中的一个索引来实现的。这样保证了在运行时进行虚函数的调用操作会快得难以置信——执行虚函数的调用操作和调用正常的函数在性能上只有很小的差别。

你可以把编译器生成的代码看成是这样的（当然，我杜撰了调用的语法）：

```
call drawables[ i ]->vtable[ 0 ];
```

另一方面，使用虚函数也存在着一个真实的弊端。你的对象大约需要为每个继承的接口持有一个虚表。这意味着每个虚的接口都会将对象的大小扩大几字节。在现实中，只有代码中出现了大量的对象，同时这些对象中的成员变量又很少时，虚函数才会带来性能上的问题。

26.2 问答题

(1) 父类的析构函数在什么时候运行？

A. 只有对一个指向父类的指针调用delete来销毁对象的时候
B. 在子类的析构函数被调用之前
C. 在子类的析构函数被调用之后
D. 在子类的析构函数被调用的时候

(2) 给定下列的类层级，在Cat的构造函数中你需要做什么？

```
class Mammal {
public:
    Mammal (const string& species_name);
};
class Cat : public Mammal
{
public:
    Cat();
};
```

A. 没什么特别要做的
B. 使用初始化列表来调用Mammal的构造函数同时带上参数 "cat"
C. 在Cat的构造函数中调用Mammal的构造函数，带上参数 "cat"
D. 你应当删除Cat的构造函数并且使用默认的版本，默认构造函数会为你解决这个问题

(3) 下面的这个类定义哪里错了？

```
class Nameable
{
    virtual string getName();
};
```

A. 它没有把getName方法设为 public
B. 它没有虚的析构函数
C. 它没有getName方法的实现，但是又没有把getName声明为纯虚的
D. 上面说得都对

(4) 在一个接口类中声明一个虚方法时，另外的一个函数需要怎么做才可以使用这个接口方法来在子类上调用方法？

A. 把接口当做一个指针参数（或者一个引用参数）
B. 什么都不用做，它会自动复制对象
C. 它需要知道被调用方法所在子类的名字
D. 我迷惑了！虚方法是什么？

(5) 继承是如何改善重用的？

A. 通过允许代码从父类继承方法
B. 通过允许父类为子类实现虚方法
C. 通过允许代码期待接收一个接口，而不是一个具体的类，允许新的类来实现接口同时保持旧的代码可用
D. 通过允许新的类继承一个具体的类的特性，这些特性可以为虚方法所使用

(6) 下列关于类的访问级别哪个是正确的?

A. 子类只能访问父类的 public 方法和数据
B. 子类能够访问父类的 private 方法和数据
C. 子类只能访问父类的 protected 方法和数据
D. 子类可以访问父类的 protected 或者 public 方法和数据

26.3　实践题

(1) 实现一个排序函数，该函数接收一个存放着指向一个接口类Comparable的指针的vector，Comparable定义了一个compare(Comparable& other)方法，如果两个对象一样则返回0，对象大于另外一个则返回1，小于另一个则返回-1。创建一个类实现这个接口，创建几个实例然后给它们排序。如果你在找关于创造什么的灵感——试试写个HighScoreElement类，它包含一个名称和一个分数，然后进行排序，这样最高分会排在第一个，但是如果两个分数相同，它们就接着按名称来排序。

(2) 为你的排序函数提供另一个实现，这次接收一个叫做Comparator的接口，其中有个方法compare(const string& lhs, const string& rhs)和之前的比较方法遵循相似的规则：如果两个值相同就返回0，lhs > rhs就返回1，lhs < rhs就返回-1。写两个不同的类来做比较：一个进行大小写的比较，一个根据字母从后往前的顺序排序。

(3) 实现一个日志方法，一个接口类StringConvertable含有一个将对象转换成一个表示自身的字符串的toString方法。日志方法应该同时也能够输出数据和时间。(这里可以找到获取类的数据的相关信息：http://www.cplusplus.com/reference/clibrary/ctime/。)再次注意我们是如何通过简单地实现一个接口来重用日志方法的。

第 27 章

命名空间

开始创建越来越多的类时，你也许要疑惑了："难道没有人已经写过实现这个功能的代码了吗？如果有，我可以拿来用吗？"有时，确实会有人已经实现过。很多核心算法和数据结构，像链表或者二叉树，已经有很稳定的、可重用的实现，而且你会需要使用那些代码。但是如果使用别人写的代码，你得注意避免命名冲突。

举个例子，你可能需要写一个叫做LinkedList的类来存储链表的实现。但是存在这样的可能，就是你使用的代码中已经有个类叫同样的名字，但是具体实现跟你的不一样。两者必有所取舍——你不能有两个类叫同样的名字。

要避免这个冲突，你可以通过创建一个*命名空间*来扩展类型的基本名称。举个例子，我可以把我的链表类放到一个叫做com::cprogramming的命名空间中去，这样我这个类型的完整的标准名称就是com::cprogramming::LinkedList。使用命名空间从根本上减少了命名冲突的几率。这里的操作符::和之前用来访问类的静态成员或者声明一个方法时的::是一样的，但是这里它不是用来访问类的元素，而是用来访问一个命名空间中的元素。

现在你可能又要疑惑了，如果命名空间真的这么好，为什么标准库的代码不使用呢？难道我们只是敲了很多没用的东西吗？

结果是你已经见过命名空间了。在每个程序的顶上都有：

```
using namespace std;
```

这样在引用像cin或者cout这些对象的时候可以避免使用完整的名称。如果不写这句声明，在每次使用那些对象的时候我们都要写std::cin或者std::cout。这个技巧在不需要用命名空间来避免某个文件中命名冲突时仍然有用，这时它可以提供一个便捷的方式让你知道文件中没有命名冲突。当文件中有命名冲突的时候，你要做的只是省略命名空间的使用声明然后把文件中的每个类型写成完整的形式就行了。

来看看如何把它用在之前的那个例子上。如果有两个不同的类都叫LinkedList，大部分文件都会在开始的时候使用命名空间com::cprogramming。如果某个文件中名称之间有了冲突，我们修改那个文件让它以com::cprogramming::LinkedList的方式来引用我的LinkedList类。我不需要修

改所有的代码，我只要修改需要同时用到两种LinkedList的地方代码。在那些文件中，我要使用完整的名称并且把使用using namespace com::cprogramming命名空间的声明去掉。

下面是个例子，你可以看看怎么把一些代码声明为某个命名空间的一部分——这里只有一个单独的变量：

```
namespace cprogramming
{
    int x;
} // <-- 注意这里不需要分号
```

现在必须以cprogramming::x来引用x或者：

```
using namespace cprogramming;
```

这样在使用了命名空间cprogramming的文件中只要写x就行了。

还可以嵌套命名空间，把一个放在另一个里面。如果是在一家大公司，有着很多不同的单元，每个单元都要做各自不同的开发，这时候你就可能会用到嵌套命名空间。在那样的情况下，你可能要使用公司的名字作为外部命名空间，然后公司内部的每个小组各自使用一个内部的命名空间。

下面是个声明嵌套命名空间的例子：

```
namespace com {
namespace cprogramming
{
    int x;
} }
```

现在x的全名就是com::cprogramming::x。（这个例子中，我没有每个命名空间都缩进——如果使用多个命名空间同时又每个都缩进的话，那缩进就会乱得难以控制了！）

你这样写：

```
using namespace com::cprogramming;
```

来访问该命名空间中的元素。

命名空间是“开放的”，也就是说可以把处于不同文件中的代码放到同一个命名空间中。举个例子，如果写了个头文件来放一个类，同时把那个类又放到了一个命名空间中：

```
namespace com {
namespace cprogramming
{
class MyClass
{
public:
    MyClass ();
```

```
};
}  }
```

在对应的源文件中，你可以这样写：

```
#include "MyClass.h"

namespace com {
namespace cprogramming
{
    MyClass::MyClass ()
    {}
}  }
```

两个文件都可以在命名空间中添加代码。你想怎么加就怎么加。

什么时候需要写**using namespace**

通常，你应该只把使用声明（using namespace）放在cpp文件中，不要放在头文件中。问题在于每个使用头文件的文件都会受到命名冲突的影响，而每个独立的cpp文件就可以控制它所使用的命名空间。一般来说，我建议在头文件中使用完整的名称然后只在cpp文件中包含使用命名空间的声明。

对于这一规则也存在着一些广为人知的例外。标准库自己事实上就违背了它，虽然是因为它有一个说得过去的理由。

如果这样写：

```
#include <iostream.h>
```

而不是这样：

```
#include <iostream>
```

那么你就不需要再包含一个 std 的使用声明了。原因是iostream.h的内容基本上就是：

```
#include <iostream>
using namespace std;
```

这是为了兼容在命名空间还没有添加到 C++ 语言之前写的那些程序，所以如果你有个这样的程序：

```
#include <iostream.h>

int main ()
{
    cout << "Hello world";
}
```

示例代码61：iostream_h.cpp

这段代码在命名空间被加入到标准库之后仍然可以编译成功。

对于新写的代码，我推荐使用新的头文件（没有.h的）这样就不会有命名空间的污染。在每个文件中加入一句`using namespace std;`也不会花太多的时间，而且这样可以让你使用“最新的”C++。

在什么情况下需要创建一个命名空间

一般情况下，如果你要处理的程序仅有几个文件，那么创建自己的命名空间可能就没什么必要的。命名空间实际上是为了在你开始创建有几十个或者数百个处于不同目录下的文件，并且确实已经能看到有命名冲突的时候使用的。简单的单个文件或者几个文件的程序真的不需要有自己的命名空间。我建议你在觉得以后会重用到代码或者程序已经大到需要拆分到不同的目录下的时候再开始把代码放到命名空间中。任何时候代码达到了这种复杂程度，你都应该使用所有可以利用的工具来保证它的条理性。

尽管命名空间在你所学到的C++特性中几乎是无关紧要的一个，但它们在处理大规模代码库的时候会派上用场。理解命名空间的作用，以及别人为什么使用它们，这些会帮助你把他人的代码整合到自己的代码中。

27.1　问答题

(1) 什么情况下需要使用`using namespace`指令？

A. 在所有头文件中，紧跟着`include`指令后面
B. 根本不能用，它们是危险的
C. 可以用在任何没有命名空间冲突的cpp文件顶端
D. 在你使用来自那个命名空间的变量之前

(2) 我们何以需要命名空间呢？

A. 为了给编译器的作者增加一些有趣的工作
B. 为代码提供更好的封装
C. 为了阻止大规模代码库中的命名冲突
D. 为了帮助阐明一个类的作用

(3) 在什么情况下应当把代码放到命名空间中？

A. 代码什么时候都应该放在命名空间中
B. 当你在开发一个有超过数十个文件的大规模程序的时候

C. 在你开发一个用来与别人共享的函数库的时候
D. B和C都正确

(4) 为什么不能把使用命名空间的声明放在头文件中?

A. 这么做是非法的
B. 没有理由不放在头文件中，使用的声明只有在头文件中才是合法的
C. 这么做会把使用声明强加给任何包含了这个头文件的人，即使这样会导致冲突
D. 如果多个头文件包含了使用声明就会导致冲突

27.2 实践题

把你在第 24 章结尾的实践题中实现的 vector 拿出来，然后把它添加到一个命名空间中。

第 28 章

文件I/O

28

文件如同计算机的命脉——如果没有文件，计算机做的任何工作最终都只能是暂时的，只能持续到用户重启计算机之前，或者应用程序运行终止的时候。C++天生就具有读写文件的能力。对文件的操作称为文件I/O（I/O表示输入和输出）。

28.1　文件 I/O 基础

文件的读写看上去很像使用cout和cin那样。与全局变量cin和cout不同的地方是你必须要声明自己的对象来读写文件[①]。这就意味着你需要知道具体的数据类型。

操作文件的两种数据类型是ifstream和ofstream。这两个名称的意思是文件输入流和文件输出流。流就是一串你可以读取或者写入的数据。这两个类型所做的工作就是接收一个文件，然后把它转换成一个可以访问的长数据流，就像是你在与用户进行交互一样。使用它们都需要fstream头文件（fstream代表文件流）。

读取文件

先来讨论如何读取文件。要读取一个文件，我们会使用到ifstream类型。可以带着一个想要读取的文件名来初始化一个ifsteam实例：

```
#include <fstream>

using namespace std;
int main ()
{
    ifstream file_reader( "myfile.txt" );
}
```

示例代码62：ifstream.cpp

这段小程序会去尝试打开myfile.txt文件，它会在程序运行的目录（这个目录叫做程序的工作

① 为了方便起见，我有时称它们为函数，但是它们确实是对象，我们会去调用它们的方法。

目录）下寻找myfile.txt文件。如果愿意的话，你也可以给定一个完整的路径，如c:\myfile.txt。

注意我说的是这段程序尝试去打开文件。它所要打开的文件可能并不存在。你可以通过调用is_open方法来检查创建的ifstream是否真的打开了一个文件，is_open方法表示ifstream对象是否成功地打开了一个文件[①]：

```
#include <fstream>
#include <iostream>


using namespace std;
int main ()
{
    ifstream file_reader( "myfile.txt" );
    if ( ! file_reader.is_open() )
    {
        cout << "Could not open file!" << '\n';
    }
}
```

示例代码63：ifstream_error_checking.cpp

在操作文件的时候，你必须要写代码来处理可能存在的失败情况，别无选择。文件可能不存在，或者已经被损坏，又或者正在被系统中的另一个进程使用。在上述这些情况下，某些文件操作可能会失败。无论何时，只要进行文件操作，你都需要做好失败的准备——磁盘访问失败，文件是损坏的，突然断电，硬盘分区坏死，所有这些都会导致文件操作失败。

文件一旦打开了，你就可以像使用cin一样来使用一个ifstream。下面的代码从一个文本文件中读取一个数字：

```
#include <fstream>
#include <iostream>


using namespace std;
int main ()
{
    ifstream file_reader( "myfile.txt" );
    if ( ! file_reader.is_open() )
    {
        cout << "Could not open file!" << '\n';
    }
    int number;
    file_reader >> number;
}
```

示例代码64：read_file.cpp

① 你可以在下面的网站上找到更多关于这些标准函数的信息，像http://en.cppreference.com/w/cpp或者http://cplusplus.com/reference/。

就像它在读取用户的输入一样，这行代码会一直从文件中读取数字，直到它发现一个空格或者别的分隔符。举个例子，如果文件中有这样的文本：

```
12 a b c
```

那么number变量在程序运行起来之后就会存储12。

由于是在操作文件，我们需要知道是否有错误发生。在C++中，检查你是否已经成功地读取到了一个值的方式是去检查执行读取操作的函数的返回值。可以像这样做：

```
#include <fstream>
#include <iostream>

using namespace std;
int main ()
{
    ifstream file_reader( "myfile.txt" );
    if ( ! file_reader.is_open() )
    {
        cout << "Could not open file!" << '\n';
    }
    int number;
    // 就是在这里检查是否成功读取了一个整数
    if ( file_reader >> number )
    {
        cout << "The value is: " << number;
    }
}
```

示例代码65：read_error_checking.cpp

通过检查调用`file_reader >> number`的结果，我们会发现读取磁盘介质时产生的问题以及所读取的数据格式导致的问题。记得前面本书开始的时候我们讨论过当想要一个数字的时候用户却输入了一个字母的情况吗？这就是防止那类问题的方法。检查输入例程的返回值，如果返回`true`，那么一切OK你可以信任所读取到的数据；如果返回`false`，那么就是某个地方出现异常了你需要把它当做错误来处理。

28.2　文件格式

向用户请求输入的时候，你可以告诉用户你想要什么，如果用户给出了错误的输入你可以提供引导，告诉用户怎么修正它。当从一个文件中读取数据的时候，你可就没有这么舒服的享受了。文件都是已经写好了的，甚至可能在你写程序之前就已经存在了。要把数据读过来你就需要知道*文件格式*。文件的格式就是文件的布局，虽然它没有必要弄得很复杂。举个例子，假设有个高分列表，你想要在程序一次一次的运行之间保存它。简单的文件格式可能就是含有 10 行，每行有个单独的数字。

一个简单的高分列表可能看上去是这样的：

```
1000
987
864
766
744
500
453
321
201
98
5
```
示例文件1：highscores.txt

你可以写个程序来读取这个高分列表：

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;
int main ()
{
    ifstream file_reader( "highscores.txt" );
    if ( ! file_reader.is_open() )
    {
        cout << "Could not open file!" << '\n';
    }
    vector<int> scores;
    for ( int i = 0; i < 10; i++ )
    {
        int score;
        file_reader >> score;
        scores.push_back( score );
    }
}
```
示例代码66：highscore.cpp

这段代码很简单——它就是打开文件然后一次读入一个分数——实际上，它都不需要依赖被换行符分开的分数——它连空格都可以处理。但是这是个实现过程中的意外，不是文件格式的特性。别的处理文件格式的程序可能就不会这么宽容地对待它们所要读入的东西了。处理文件格式有个好的原则叫做Postel法则，就是“宽进严出”。换句话说，生成文件的代码应当小心谨慎地遵循规格说明，但是读取文件格式的代码应当足够强壮来抵抗那些由不是特别优秀的代码所造成的错误。在上面的示例程序中，我们在接收换行分隔符的同时宽容地接收了空格分隔符。

文件的结束（EOF）

上面这段代码是遵循一个很特别的文件格式而写的，而且你会注意到它根本没有尝试进行错误处理。比如，假使没有10个条目怎么办，这段代码不会停止读取文件，哪怕它已经读到了文件

的最后。打个比方，假如游戏才只被玩过两次，它就还没有10个分数记录，文件中也就没有10个条目。EOF这个词常用来表示已经到达了文件的最后的状态。

我们可以通过处理文件不足10个条目的情况，来让代码变得健壮（对接收的数据要求宽泛）。通过再一次地检查用来读取输入的方法的返回值，我们就可以处理不足10条的情况了。

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
    ifstream file_reader( "myfile.txt" );
    if ( ! file_reader.is_open() )
    {
        cout << "Could not open file!" << '\n';
    }
    vector<int> scores;
    for ( int i = 0; i < 10; i++ )
    {
        int score;
        if ( ! file_reader >> score )
        {
            break;
        }
        scores.push_back( score );
    }
}
```

示例代码67：highscore_eof.cpp

当这段代码处理的文件不足10个条目的时候，它在读到文件末尾的时候会立即停止。通过使用vector而不是用固定长度的数组，我们可以轻松地处理短一点的文件。vector会准确地存储着读进来的东西，没有别的。如果用数组来完成了同样的工作，我们得需要有变量来存储数组中条目的数量——我们不能假定整个数组都被存储满了。

有些情况下你操作文件的时候，会想要把文件中所有的数据都读进来直到文件结束。在这样的情况下，你需要能够辨别出由于到达文件末尾而导致的读入失败和由于文件中的错误而导致的读入失败。`eof`方法会指示出是否到达了文件的末尾。你可以写个循环来不停地读入数据，检查每次读入的结果，直到出现读入失败。接着你可以检测`eof`是不是返回`true`；如果是的，那么已经读到文件结尾了；如果不是，那就是文件有问题。你可以通过调用`fail`方法来检查别的原因导致的失败，如果有非法的输入或者从设备中读取的时候出了问题就会返回`true`。一旦读到了文件的最后，你必须调用`clear`方法以便执行进一步的文件操作。我们很快会见到一个使用所有这些方法的例子，就在下面这部分，把一个新的分数写到高分列表。

读取文件和与用户交互还有另一个重要的区别。如果我们改一下高分列表，在分数的基础上

再加上玩家的名字会怎样呢？我读取分数时也要读取玩家的名字——我们得改改代码来处理这个问题。我们老版本的程序将会无法读取新的文件格式。如果你有很多的用户并且你想要修改文件的格式，那么这就成为主要的麻烦了。有一些技巧可以为文件格式提供*前瞻性*，可以添加一些可选的字段或者给老版本的程序加上忽略文件格式中新元素的功能。但是这些技巧都超出了本书的范围。现在而言，只要明白定义一个文件格式（在某些方面）比定义一个基本接口更加需要慎重。

28.3　写文件

我们写文件要用的数据类型叫做ofstream，表示文件输出系统。这个类型和ifstream几乎是一样的，除了你要像使用cout那样来用它，而不是像使用cin那样。

我们来看个简单的程序，将0~9的值写出到叫做highscores.txt的文件中（我们很快会让这段代码能制造出像一个高分列表的东西）。

```
#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main ()
{
    ofstream file_writer( "highscores.txt" );
    if ( ! file_writer.is_open() )
    {
        cout << "Could not open file!" << '\n';
        return 0;
    }

    // 由于没有任何真实的分数，接下来输出数字10~1
    for ( int i = 0; i < 10; i++ )
    {
        file_writer << 10 - i << '\n';
    }
}
```
示例代码68：ofstream.cpp

这段代码中就无需担心到达文件末尾的问题了。当你向文件中写入并且写到文件的末尾了，ofstream会为你扩展文件。这叫做向文件进行*添加*。

新建文件

当你使用ofstream写入一个文件时，默认情况下，如果文件不存在它会创建一个，或者在文件存在的情况下会重写文件。如果是在保存一个高分列表，你可能不会介意每次去重写文件因

为你会写入所有的数据。但是，如果在维护一个运行日志——比如要保存每次用户打开程序的日期和时间——你肯定不想每次写入都覆盖你的日志。

ofstream构造函数接收第二个参数，它指定了文件应该被如何处理：

ios::app	在文件后面作添加，每次写入之后把位置设到最后
ios::ate	把当前位置设为最后
ios::trunc	删除文件中所有的东西（截去文件）
ios::out	允许向文件输入
ios::binary	允许对流进行二进制操作（读取文件时同样可以这样）

如果要选择多个选项，比如打开一个文件来添加内容并且使用二进制IO（很快会讲到），你可以用管道（ | ）把这些操作结合起来[①]：

```
ofstream a_file( "test.txt", ios::app | ios::binary );
```

这段代码打开文件而不毁坏文件当前的内容，允许把二进制数据写到文件的末尾。

28.4　文件位置

当程序读入一个文件（或者写入一个文件）时，文件I/O的代码需要知道读或者写发生在什么地方。把它想作屏幕上的光标，它会告诉你下一个输入的字母会出现在什么地方。

对于基本的操作无需担心位置问题——你可以写代码去读取文件的任何地方，或者将数据写入到文件的任何地方。然而你可以在不做读取操作的情况下在文件中改变位置。在处理存储着复杂数据，如ZIP文件或者PDF文件，或者你有一个庞大的文件，读取每个字节都会很慢或者不可能读取到每个字节（假设你在实现一个数据库），这时候移动在文件中的读取位置就很重要了。

事实上文件中两个不同的位置——一个代表程序下一个要读取的地方，一个代表着程序下一个要写入的地方。你可以使用tellg和tellp方法来获取当前的位置。这两个方法给你返回当前读取（g代表get）和写入（p代表put）的位置。

你也可以在当前位置的基础上移动来设置你在文件中的位置，使用seekp和seekg。你可能已经从名字上猜到了，在文件中移动叫做seeking。当在一个文件中搜寻的时候，你会把读的位置或者写位置移动到一个新的地方。这两个方法接收两个参数，一个是搜寻的距离，一个是搜寻操作的源头。搜寻的距离是用字节来度量的，而源头则不是你当前的位置、文件的开头就是文件的结尾。在搜寻操作之后，你将可以在文件中新的位置开始读取（或者写入）。通过搜寻来改变一

① 管道符号是位运算符——或。每个ios::操作都会设置一个位为true，你可以使用位或来组合这些操作。更多关于位运算符的内容请参考 http://www.cprogramming.com/tutorial/bitwise_operators.html。

个位置对别的位置不会产生影响。

文件中位置的三个标志位：

ios_base::beg	从文件的开始的地方进行搜寻
ios_base::cur	从当前位置开始搜寻
ios_base::end	从文件的末尾开始搜寻

举个例子，要在开始写入之前移动到文件开始的地方，可以这样写：

```
file_writer.seekp( 0, ios_base::beg );
```

tellp和tellg的返回值是标准库中定义的一个特殊的类型叫做streampos。它可以与整型进行相互之间的转换，但是使用streampos，我们能够更加明确地表示该数据的类型。整型可以在任何地方使用，但是streampos意味着有个特殊的目的。一个streampos类型的变量可以存储文件中的位置和用来搜寻到那些位置。在我们的代码中使用正确的变量类型可以让变量的作用清楚明了。

```
streampos pos = file_reader.tellg();
```

在有些情况下，你不会需要在文件中进行搜寻——将一个文件从开始读到最后就足够了。然而，很多文件的格式为了可以向文件中添加新的数据而做了优化。当你向文件中添加新数据的时候，在文件末尾添加会比插入到文件的中间位置快很多。向文件中间插入的问题就在于你必须要移动文件中插入点之后所有的东西——就像在数组的中间插入一个元素一样[①]。

来修改之前的读取高分的程序，让它可以在文件中添加新的高分。为了让它更有趣，我们会把值插入到文件中正确的地方。

要实现这个，我们需要能够读和写文件，所以将会使用fstream类，它同时允许读和写的操作。就把它想作ofstream和ifstream缠绵在一起吧。首先我们将从用户那里读入一个新的高分，然后会读入文件中的每一行，直到发现一个低于前面输入的分数。这儿就是要插入新分数的地方。我们会保存这个位置，将文件中剩余的行都读入到一个vector中，然后再回到这个地址。写出新的分数，接着再把剩余的分数写回到文件中，替换掉原先在那里的一行行数据。

由于使用的是fstream，我们会得到能够同时读和写的所有好处，但是现在需要明确地告诉构造函数同时以读和写的目的去打开文件。我们会使用标志位ios::in|ios::out标明。在运行程序之前需要创建一个高分列表文件；这里的程序不会为你创建一个空文件。

```
#include <fstream>
#include <iostream>
```

① 有一个特殊情况：如果相同长度的新数据来覆盖已有的数据，你就不需要移动什么，这就和写在文件的末尾速度一样快。

```
#include <vector>

using namespace std;

int main ()
{
    fstream file ( "highscores.txt", ios::in | ios::out );
    if ( ! file.is_open() )
    {
        cout << "Could not open file!" << '\n';
        return 0;
    }
    int new_high_score;
    cout << "Enter a new high score: ";
    cin >> new_high_score;

    // 下面的while循环在文件搜索直到发现一个比当高分小的值；这时候
    // 我们就知道在这个值之前插入高分。为了确保知道正确的位置，
    // 记录下在当前分数之前的位置，也就是pre_score_pos
    streampos pre_score_pos = file.tellg();
    int cur_score;
    while ( file >> cur_score )
    {
        if ( cur_score < new_high_score )
        {
            break;
        }
        pre_score_pos = file.tellg();
    }

    // 如果返回失败，而又不是在文件的末尾，那就是读入有问题
    if ( file.fail() && ! file.eof() )
    {
        cout << "Bad score/read--exiting";
        return 0;
    }

    // 如果不调用clear，在遇到EOF的时候就不能再向文件中写入数据
    file.clear();

    // 回到上一次读取的前面一个位置，来进行读取，这样就可以读入
    // 所有比我们高分低的那些分数，然后把它们在文件中往后移一个位置
    file.seekg( pre_score_pos );

    // 现在将读取所有的分数，从之前读入的那个开始
    vector<int> scores;
    while ( file >> cur_score )
    {
        scores.push_back( cur_score );
    }
    // 我们准备在这个循环中读到文件的结尾，因为想要读入文件中
    // 所有的分数
    if ( ! file.eof() )
    {
```

```
        cout << "Bad score/read--exiting";
        return 0;
    }
    // 由于遇到了EOF，需要再次清理一下文件，这样我们可以进行写操作
    file.clear();

    // 回到想要进行插入操作的位置
    file.seekp( pre_score_pos );
    // 如果不是写入到文件的开始处，我们就需要包含进来一个新换行
    // 原因是当一个数字读进来的时候，程序会再第一个遇到的空格处停下来，
    // 这样的话在写之前所处的位置是在前面那个数字的末尾，
    // 而不是第二行的开头
    if ( pre_score_pos != 0 )
    {
        file << endl;

    // 写出我们新的高分
    file << new_high_score << endl;
    // 遍历剩下的分数，把它们都写到文件中
    for ( vector<int>::iterator itr = scores.begin(); itr != scores.end(); ++itr )
    {
        file << *itr << endl;
    }
}
```

示例代码69：file_position.cpp

28.5 接受命令行参数

当写文件交互的程序时，你通常想让用户提供文件名作为命令行的一个参数。这么做通常会让程序更好用，并且让写脚本来调用你的程序也变得更简单。我们先短暂地暂停一下研究文件的读写，这样就可以利用命令行参数的特性让程序更漂亮。

命令行参数在程序的名称之后给出并且是由操作系统传递给程序：

```
C:\my_program\my_program.exe arg1 arg2
```

命令行参数直接被传递到主函数中——要使用命令行参数，你必须提供完整的主函数声明（之前我们看到的所有主函数都只有空的参数列表）。实际上，主函数接收两个参数：一个参数是命令行参数的个数，另一个参数是所有命令行参数的一个完整列表。

主函数完整的声明像这样：

```
int main (int argc, char *argv[])
```

整型的argc是参数的个数。它是从命令行传递给程序的参数个数，包括了程序名称。你也许想知道为什么不需要在每个程序中都包含这些参数；答案很简单，如果你不把它们加进来，编译器就会忽略它们被传递到程序中的事实。

字符指针数组是所有参数的列表。argv[0]是程序的名称，或者是个空字符串如果程序名不可用的话。在它之后，每个比arc小的元素都是个命令行参数。你可以把每个argv元素就像字符串一样使用。argv[argc]是个空指针。

来看一个示例程序，它接收一个命令行参数——在这个例子中，程序接收一个文件名然后把整个的文本输出到屏幕上。

```
#include <fstream>
#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
    // 为了程序正确执行argc应当为2，参数有程序名和文件名

    if ( argc != 2 )
    {
        // 在输出用途说明的时候，你可以使用argv[ 0 ]作为文件名
        cout << "usage: " << argv[ 0 ] << " <filename>" << endl;
    }
    else
    {
        // 我们假设argv[ 1 ]是个要打开的文件名
        ifstream the_file( argv[ 1 ] );
        // 不要忘记检查文件是否成功打开
        if ( ! the_file.is_open() )
        {
            cout << "Could not open file " << argv[ 1 ] << endl;
            return 1;
        }
        char x
        // the_file.get( x )从文件中读取下一个字符到x中，如果
        // 到达文件末尾或者有错误发生就返回 false
        while ( the_file.get( x ) )
        {
            cout << x;
        }
    } // the_file在这里被它的析构函数隐式地关闭了
}
```

示例代码70：cat.cpp

这段程序使用完整的主函数声明以便使用命令行参数。首先它会检查确保用户提供了一个文件名。然后程序试着打开它来看看文件是否合法。如果文件是合法的，那么它就是被打开的——如果不是，程序向用户报告一个错误。如果文件成功打开了，那么它就会把文件的每个字符输出到屏幕上。

处理数字命令行参数

如果希望接收一个命令行参数并且把它作为一个数字来使用，你可以通过把它作为一个字符串读入接着调用atoi函数（atoi代表 ASCII 转换到整型）。atoi函数接收一个char*然后返回该字符串所表示的整型，要使用它你必须包含cstdlib头文件。举个例子，下面的程序读入一个命令行参数，把它转换成一个数字，并且输出那个数字的平方：

```
#include <cstdlib>
#include <iostream>

using namespace std;
int main (int argc, char *argv[])
{
    if ( argc != 2 )
    {
        // 在输出使用说明的时候，你可以使用argv[ 0 ]作为文件名
        cout << "usage: " << argv[ 0 ] << " <number>" << endl;
    }
    else
    {
        int val=ato;(argv[1]);
        cout << val * val;
    }
    return 0;
}
```

示例代码71：atoi.cpp

28.6 二进制文件 I/O

到目前为止我们已经学过如何去处理含有文本数据的文件；现在把注意力转向处理二进制文件，我们经常为了追求最高效率而去使用二进制文件。二进制文件需要不同于文本文件的编程技术。现在，不要被迷惑——系统中任何一个文件都是以二进制的形式存储的。但是在很多情况下，文件是以一种用户可以阅读的方式来写入的。举个例子，C++源文件全部都是由基本的编辑器就可以阅读的字符组成的。这种文件，它的每个字节都是可阅读字符的一部分，这就叫做文本文件。

然而，不是所有的文件都仅仅包含文本。有些文件是由无法输出字符的字节组成的。取而代之的是，这些文件只是由一个或者多个结构体直接写到磁盘上的二进制数据。

举个例子，假设有个代表运动员的结构体：

```
struct player
{
    int age;
    int high_score;
    string name;
};
```

如果要把这个结构体写入到一个文件中，关于如何去做你有两个选择。首先，可以以文本字段的形式来记录年龄和最高分，把它们和姓名放在一起，这样文件就可以在记事本中打开。它看上去可能像这样：

```
19
120000
Tom
```

这种表示方法使用了6个字符来代表最高分。我们已经学过，一个字符需要1字节来存储，这就意味着存储最高分会占用6字节。但是最高分是个整型，而一个整型通常只有4字节（在32位操作系统上），所以不是应该只用4字节来存储它吗？你说对了！但是如果只使用4字节去写数字，我们就不能在文本编辑器中打开这个文件去看它到底是什么数字了。为什么呢？因为在以字符串的形式把120000写入文件的时候，它通过编码使每个字符占用1字节来存储实际的数字为字符的形式。在你把数字直接写入文件时，那些字节根本没有被编码成字符。所以你现在有4字节组成的整型写入到文件中了。如果文本编辑器去读取这个文件，它会把那4字节当做4字符来对待，但是它输出的字符和我们所要展示的数字没有任何关系！打开的结果也会毫无意义因为我们是在以不同的方式为文件编码。

二进制文件格式占用更少的空间。在上面的例子中，我们看到以字符存储120000比使用二进制表示多占用50%的空间。你能够想象到，如果在通过网络传输数据或者硬盘不是很快或者足够大的话，这会产生很大的影响。另一方面，二进制文件不易于阅读和理解——你不能简单地在文本编辑器中打开一个二进制文件去看它里面是什么数据。文件格式的设计者面临着创造高效的格式与创建任何人都可以理解并易于修改的文件格式，要这两者之间取平衡点。基于文本的标记性语言如XML通常用来创建占用更多空间，但是非常易于人们理解的文件格式。

在存储空间有限，处理器足够快速时，可以使用像 ZIP 这样的压缩技术来减少存储文本文件所需要的空间。由于解压一个文件很容易，这些文件仍然是方便处理的，同时比没有压缩过的文本文件小了很多。

尽管如此，二进制文件还是很常见的——很多已有的文件格式就是二进制的，而且很多文件格式真的必须是二进制的——任何存储图像，视频或者音频的文件都不具备有意义的，精确的文本表示。而且在需要追求最大性能或者节省空间时，二进制文件仍然会胜出——举个例子，在Office 2007中微软引入了新的文件格式，基于ZIP内部的XML文件。但是他们也在Excel（.xlsb）中添加了一种二进制格式，为了方便那些追求最大性能的用户。换句话说，二进制文件就在这里，在任何需要设计一个文件格式的时候，你都必须评估更简单的实现和表示（基于文本的格式）与性能和大小（二进制格式）之间的平衡。

所以，你也许会问，到底怎么去处理一个二进制文件呢？

28.6.1 处理二进制文件

第一步就是以二进制模式打开一个文件：

```
ofstream a_file( "test.bin", ios::binary );
```

一旦文件打开了，你不能使用前面用的输入输出函数——要使用专门处理二进制数据的函数。我们需要直接把一块内存中的字节写入到文件中。我们将要使用的方法叫做write，它接收一个指向一块内存的指针和要写入到文件中的内存的大小。指针的类型是char*，但是你的数据不必须是字符。那么，为什么使用字符呢？在C++中，处理单独字节的方式是使用一个byte变量，也就是char，或者一个指向一系列byte的指针，也就是char*。想要把一系列文字的字节写入到文件中时，你需要提供一个char*把单独的字节放入文件中。为了把一个整型写入到文件中，你得把它当做一系列字节，也就是char*，并且把这个指针传递给将来自内存的字节直接写入到文件中的方法。为了实现这个功能，write方法会把字符一个一个写出，每个字节，按顺序一个接一个。例如，假设你有个数字255。在内存中，它会以字节0xFF来表示（十六进制的255）。如果有个整型变量存储着0xFF的字节，在内存中它会是这样的：

```
0x000000FF
```

或者，一字节一字节地：

```
00 00 00 FF
```

要把一个整型写入文件中，我们需要一个直接引用这一系列字节的方式。这就是使用一个char*的原因：这不是因为它可以代表ASCII；是因为它可以处理字节。

我们还需要有一个方式来告诉编译器它应该像对待一个字符数组一样对待我们的数据。

28.6.2 转换到**char***

那么我们怎么告诉编译器把一个变量当做指向字符的指针，而不是指向它真正类型的指针呢？要求编译器以不同的类型来处理一个变量叫做*类型转换*。类型转换告诉编译器——“不，说真的，我知道自己在干什么；我真的想以这种方式使用这个变量”。我们想要把一个变量当做一系列单独的字节来处理，所以需要使用一个转换来强制编译器支持访问单个字节。

两个最基本的类型转换是static_cast和reinterpret_cast。static_cast是你想要在相关的类型之间做转换时使用的——举个例子，告诉编译器把双精度型当做整型来处理这样你就可以截取它——比如static_cast<int>(3.4)。要被转换成的类型在尖括号内给出，跟在转换方法的名字后面。

尽管在这个例子中，我们想要完全忽略类型系统并且让编译器属于一个完全不相干的类型来重新解释一系列的字节。要完成这个操作，我们需要reinterpret_cast。举个例子，把一个整型数组当做字符数组来用，可以这么写：

```
int x[ 10 ];
reinterpret_cast<char*>( x );
```

顺便说一下，处理二进制数据是少数几个合理使用reinterpret_cast的地方。无论何时你见到reinterpret_cast的时候，要带着怀疑的态度！这是让编译器去做它正常情况下不做的事的一个强大的方式，并且，作为结果，编译器不会像检查别的代码那样去仔细地检查使用强制转换的代码。在这个特别的例子中，我们确实需要得到1字节序列的内存，所以这就是我们想要的；但是如果那不是你的意图，使用reinterpret_cast就不是个好的主意了。

28.6.3　二进制I/O的一个例子

最后，我们终于可以展示二进制输入输出了！这个示例代码填充一个数组然后把它写到文件中。它使用我们之前见过的write方法，接收一个char*作为源数据，还有从那个源写出的数据的大小。在这个例子中，源是数组，数组的大小就是数组的字节长度。

```
int nums[ 10 ];

for ( int i = 0; i < 10; i++ )
{
    nums[ i ] = i;
}
a_file.write( reinterpret_cast<char*>( nums ), sizeof( nums ) );
```

以一个整型数组开始，但是通过把它转换成一个char*，它会简单地被当做字节数组来处理，这个字节数组会直接被写到磁盘上。当我们之后再读入这些字节的时候，内存里就正好是同样的字节集合，并且可以把这段内存重新转换成整型来获取和原来一样的数值。

注意，要写入的大小是以sizeof操作符来提供的。sizeof命令在获取一个特定变量的大小时很有用。在这个例子中，它返回组成数组nums的所有字节数。

尽管如此，在对指针使用sizeof的时候还是要小心。当给它一个指针，它会给你指针的大小，而不是指针所指向的内存的大小。上面的代码可以正常工作是因为nums是被声明为数组而不是指针，而sizeof知道整个数组的大小。如果你有个指针变量int *p_num，这个变量的大小是（通常情况下）4字节因为保存一个地址只需要这么多。如果想得到所指向的东西的大小，你可以写sizeof(*p_num)。在这里，结果会和sizeof(int)一样。如果指针指向一个数组（如果你写了int *p_num = new int[length]），可以这样去获取总的大小：sizeof(*p_num) *length。

你还可以使用write方法直接把一个结构体写到文件中去。例如，假设有这么个结构体：

```
struct PlayerRecord
{
    int age;
    int score;
};
```

你可以简单地创建一个PlayerRecord的实例，然后把它写入到文件中：

```
PlayerRecord rec;
rec.age = 10;
rec.score = 890;

a_file.write( reinterpret_cast<char*>( & rec ), sizeof( rec ) );
```

注意，在这个示例中我们获取了rec的地址，为的是传入指向结构体的指针。

28.6.4　把类存储到文件中

如果想要在结构体中添加一个非基础的数据类型呢？举个例子，我们在结构体中加入一个字符串会怎样呢？

```
struct PlayerRecord
{
    int age;
    int score;
    string name;
};
```

在这个例子中，我们简单地以字符串的形式将运动员的名字添加到结构体中。但是如果要把它写入到文件中去，在写到字符串的时候会发生什么？它会把存储在字符串中的信息写出去——但是可能写的不是字符串本身的内容。

字符串类型是以指向一个字符串的指针来实现的（可能和一些别的数据一起，比如字符串的长度）。在我们以二进制数据的形式写出结构体的时候，它会直接写出字符串中存储的东西——指针和长度。但是指针只有程序在运行的时候才有意义！指针本身的值——内存地址——一旦程序退出以后就没用了，因为那个地址已经没有任何东西了。下一次有人读入这个结构体，它会得到一个指向没有正确分配内存的指针，或者指向与我们的字符串毫无关系的数据。

我们需要想出一个固定的，定义良好的格式来在磁盘上表示二进制数据，而不是盲目地将结构体本身直接写到磁盘上。我们的格式是写出字符串里面的字符和字符串的大小（需要大小的原因很快就会清楚）。来看看那会是什么样子。


```
PlayerRecord rec;
rec.age = 11;
rec.score = 200;
rec.name = "John";

fstream a_file( "records.bin", ios::trunc | ios::binary | ios::in | ios::out);

a_file.write( reinterpret_cast<char*>( & rec.age ), sizeof( rec.age ) );
a_file.write( reinterpret_cast<char*>( & rec.score ), sizeof( rec.score ) );
int len = rec.name.length();
a_file.write( reinterpret_cast<char*>( & len ), sizeof( len ) );
a_file.write( rec.name.c_str(), len + 1 ); // + 1 是因为空的结束符
```

首先，注意到使用c_str方法来获取内存中字符串的指针，而不是在内存中没有确定的布局的字符串对象本身。如果字符串是“abc”，那么调用c_str会给你一个带有字母“abc”的字符序列的地址。字符串会以一个值为 0 的字符结尾；这个值为 0 的 byte 数值叫做空结束符，并且它标示着字符串的结尾[①]。这种格式的字符串叫做 C 字符串，因为在 C 语言中 C 字符串是全局都可用的字符串格式。

我们把字符数据写入到二进制文件中是没有问题的；就算是在写入字符到文件中，这仍然是在写二进制数据——仅仅是碰巧二进制数据同时也是人类可读的。

我们不把开始时的结构体写出去也是完全没有问题的——重要的是可以把磁盘上的文件格式转换成内存中的一个对象，而不是直接从内存中把字节写到磁盘上。文件格式是数据的一种表示方式；结构体是数据的另外一种表示方式。两者存储着同样的数据，但是内存中的结构体的格式没有必要和文件中的数据格式一模一样。

28.6.5　读取二进制文件

要读入一个二进制文件，我们会使用适当命名的read方法。read方法的参数几乎和write方法是一样的：一个存放数据的地方以及要读取的数据量[②]。要从文件中读取一个整数，可以这样写代码：

```
int x = 3;
a_file.read( reinterpret_cast<char*>( & x ), sizeof( x ) );
```

在处理文件的时候，你需要有一些方式同时写入和读取各种想要存储在文件中的数据结构。我们来看看如何读取一个PlayerRecord。首先，我们从简单的开始，重置文件位置，接着读入直接写到磁盘而没有修改过格式的age和score字段。

```
a_file.seekg( 0, ios::beg );

PlayerRecord in_rec;

if ( ! a_file.read( reinterpret_cast<char*>( & in_rec.age ), sizeof( in_rec.age)))
{
    // 错误处理
}
if ( ! a_file.read( reinterpret_cast<char*>( & in_rec.score ), sizeof( in_rec.score ) ) )
{
    // 错误处理
}
```

① 有时你会看到空结束符写作'\0'。这完全是正确的写法。0和'\0'之间的区别就在于'\0'本来的类型就是字符，而0则是将要被转换成字符的整型。在我们这里，两者都可以。

② 有个值得注意的是传入write的指针可能是const，这意味着你可以传入一个指向待写入的const对象的指针。顺便说一下，这种情况下，你需要使用reinterpret_cast<const char*>（注意const在类型转换中）。

那么读入字符串又是什么情况呢？我们不能仅仅从文件中字符串的开头读入char*（内存中的格式和磁盘上的格式是不一样的），必须读入char*然后创建一个新的字符串。

现在知道为什么要存储字符串的长度了吧：我们需要知道存储char*要分配多少空间。我们会读入字符串的长度，然后为它分配内存，最后会把字符串读入到这段内存中。

```
int str_len;

if ( ! a_file.read( reinterpret_cast<char*>( & str_len ), sizeof( str_len ) ))
{
    // 错误处理
}
// 执行一次明智的检查来确保没有分配过多的内存!
else if ( str_len > 0 && str_len < 10000 )
{
    char *p_str_buf = new char[ str_len ];
    if ( ! a_file.read( p_str_buf, str_len + 1 ) ) // + 1 是因为null终止符
    {
        // 错误处理
    }
    // 确认字符串是null终止的
    if ( p_str_buf[ str_len ] == 0 )
    {
        in_rec.name = string( p_str_buf );
    }
    delete p_str_buf;
}

cout << in_rec.age << " " <<in_rec.score << " " << in_rec.name << endl;
```

下面这个完整的可以正常运行的程序由你来做验证：

```
#include <fstream>
#include <string>
#include <iostream>

using namespace std;

struct PlayerRecord
{
    int age;
    int score;
    string name;
};

int main ()
{
    PlayerRecord rec;
    rec.age = 11;
    rec.score = 200;
    rec.name = "John";
```

```
    fstream a_file( "records.bin", ios::trunc | ios::binary | ios::in | ios::out );

    a_file.write( reinterpret_cast<char*>( & rec.age ), sizeof( rec.age ));
    a_file.write(reinterpret_cast<char*>(  & rec.score ), sizeof( rec.score ));

    int len = rec.name.length();
    a_file.write(reinterpret_cast<char*>( & len ), sizeof( len ) );

    a_file.write( rec.name.c_str(), rec.name.length() + 1 );

    PlayerRecord in_rec;

    a_file.seekg( 0, ios::beg );
    if ( ! a_file.read( reinterpret_cast<char*>( & in_rec.age ), sizeof( in_rec.age )))
    {
        cout << "Error reading from file" << endl;
        return 1;
    }
    if ( ! a_file.read( reinterpret_cast<char*>(& in_rec.score ), sizeof( in_rec.score )))
    {
        cout << "Error reading from file" << endl;
        return 1;
    }

    int str_len;

    if ( ! a_file.read( reinterpret_cast<char*>( & str_len ), sizeof( str_len )))
    {
        cout << "Error reading from file" << endl;
        return 1;
    }

    // 执行一次明智的检查来确保没有分配过多的内存!
    if ( str_len > 0 && str_len < 10000 )
    {
        char *p_str_buf = new char[ str_len ];
        if ( ! a_file.read( p_str_buf, str_len + 1 ) ) // + 1 是因为null终止符
        {
            delete p_str_buf;
            cout << "Error reading from file" << endl;
            return 1;
        }
        // 确认字符串是null终止的
        if ( p_str_buf[ str_len ] == 0 )
        {
            in_rec.name = string( p_str_buf );
        }
        delete p_str_buf;
    }

    cout << in_rec.age << " " <<in_rec.score << " " << in_rec.name << endl;
}
```

示例代码72：binary.cpp

你运行了这段程序之后，试试在记事本或者别的文本编辑器中打开生成的文件。你可以读到姓名John，因为它是以字符串来存储的，但是除此之外的都没有意义。

28.7　问答题

(1) 哪个数据类型可以用来读取文件？

A. `ifstream`　B. `ofstream`　C. `fstream`　D. A 和 C

(2) 下列哪句是正确的？

A. 文本文件比二进制文件占用更少的内存空间
B. 二进制文件更易于调试
C. 二进制文件比文本文件更节省空间
D. 文本文件太慢了，不能在真实的程序中使用

(3) 在写入二进制文件的时候，为什么不能传入一个指向字符串对象的指针？

A. 你每次都要传入一个`char*`到`write`方法中
B. 内存中可能没有保存字符串对象
C. 我们不知道字符串对象的布局，它可能含有会被写入到文件中的指针
D. 字符串太大了必须一点一点地写入

(4) 下列关于文件格式哪个是正确的？

A. 文件格式和别的输入一样易于修改
B. 修改文件格式需要考虑旧版本的程序读取文件时会发生什么事
C. 设置文件格式时需要考虑新版本的程序打开旧版本的文件会发生什么事
D. B 和 C

28.8　实践题

(1) 重新实现插入分数到正确的位置的最高分程序，但是使用二进制文件格式而不是文本文件格式。你如何辨别程序是正常工作的呢？创建一个程序以文本文件来显示二进制文件。

(2) 修改你在第19章中实现的HTML解析器，让它能够从磁盘上的文件读取数据。

(3) 创建一个简单的XML解析器。XML是个基础的格式化语言，和HTML相似。它的文档是树形结构的节点，格式是<node>[data]</node>，[data]不是文本就是另外嵌套的节点。XML节点可能有属性，格式是<node attribute = "value"></node>。（真正的XML说明包含了更多的细节，但是那需要费很大的劲来实现。）你的解析器应当接收一个有几个方法的接口类，下列这些事发生

时它会调用这些方法。

1) 当读入节点的时候，它会带着节点的名字调用nodeStart。

2) 当读入属性的时候，它会调用attributeRead；这个方法应当总是在针对属性相关联的节点nodeStart之后立即被调用。

3) 当节点有文本正文时，调用nodeTextRead，带着文本的内容，以字符串的形式作为参数。如果你遇到像这样的情况<node>text<sub-node>text</sub-node>more text</node>，在sub-node之前和之后的文本需要分别调用nodeTextRead 。

4) 当读到end-node的时候，带着节点的名字去调用nodeEnd。

5) 你可以把任何<或>当做节点的开始。如果XML文件的作者要让<或>出现在文本中，它应当被写作<或>（意思是大于和小于）。由于符号与也是必须避免的，它们必须以&的形式出现。在代码中你无需翻译<和>或者&。

下面是一些XML示例文档让你作为输入的测试数据：

```
<address-book>
<entry>
<name>Alex Allain</name>
    <email>webmaster@cprogramming.com</email>
</entry>
<entry>
    <name>Joe Doe</name>
    <email>john@doe.com</email>
</entry>
</address-book>
```

还有：

```
<html>
    <head>
        <title>Doc title</title>
    </head>
    <body>This is a nice <a href="http://www.cprogramming.com">link</a> to
a website.</body>
</html>
```

为了测试解析器能正常工作，你可以写段代码来显示文件中每个解析出来的元素，然后认证它获取的就是你想要的元素。或者可以实现下一个习题，它会展示你的解析器在使用中的一个例子。

(4) 重写HTML解析器让它使用你的XML解析器，而不是之前的手动解析。添加对列表显示的支持。你应当能够读取
标签或者<nl>标签，来识别无序和有序的列表。各个列表项应当在
和
标签之间。显示出来的：

```
<ul>
<li>first item</li>
```

```
<li>second item</li>
</ul>
```

应该是：

```
* first item
* second item
```

对于：

```
<nl>
<li>first item</li>
<li>second item</li>
</nl>
```

则是：

```
1.first item
2.second item
```

如果有第二个序列表出现，请确保重启标序功能。

第 29 章

C++中的模板

29

到目前为止你必须要为C++中的任何东西指定类型。如何声明一个变量呢？你需要一个类型。声明一个函数——你需要给出所有参数的类型,还有返回值以及函数所有的局部变量的类型。

不过，有时候你可能想写通用的代码——使用什么类型无关紧要，因为逻辑对于所有的类型都是一样的。你已经见过别人写的这类代码的一些例子，就是STL。STL是一个以通用方式操作的数据结构（也有算法）的集合——它们可以持有任何程序员所要求的类型。往STL vector中存储条目的时候，你要告诉vector它将要存储的数据类型；不需要局限于预先定义好的类型。STL的作者写了一个vector的实现能够存储所有的数据类型。

他们是如何实现这么棒的特性的呢？原来他们使用了C++的一个叫做*模板*的特性。模板允许你写个函数或者类的“模板”，而不需要给出其中所有元素的类型；然后当需要支持一个特定类型的时候，编译器可以创建或者*初始化*一个包含所需类型的模板的版本。这就是发生在你写`vector<int> vec;`的时候，编译器用`int`类型来填充`vector`模板，创建一个可用的类。

如你已经见过的,使用模板是很简单的。本章都是关于创建自己的模板函数和模板类的内容。先来看一看模板函数。

29.1 模板函数

模板是创造更为通用的函数的完美解决方案。举个例子，你可能想要写个小的帮助函数来计算三角形的面积：

```
int triangleArea (int base, int height)
{
    return base * height * .5;
}
```

如果你要得出一个高为0.5底也是0.5的三角形面积，怎么办？由于两个参数都是整型的传入的数据会被截取成0，所以即使面积不为0，函数也会返回0。

另外一个选择是再写一个方法：

```
double triangleAreaDouble (double base, double height)
{
    return base * height * .5;
}
```

这段代码看起来和第一个函数一样……除了我们把所有的类型声明成double而不是integer的那一行。如果要对另外一个类型的参数做同样的操作——也许是个自定义的数字类——我们就得写该函数的第三种实现了。

C++模板是解决这种问题的完美方案。模板允许你把数据类型“提取出来”。函数调用者列出要使用的类型，作为交换，编译器会为每个调用者要求的类型生成一个函数。

模板声明的语法上起来有一点吓人，但是我会把它拆开来解释，这样你就明白它的意思了。下面使用模板的语法来写上面那个函数：

```
template <typename T>
T triangleArea (T base, T height)
{
    return base * height * .5;
}
```

首先，我们使用template关键字来声明这个函数为模板。接着，我们在尖括号中列出了模板参数——这些参数是模板的使用者将要给定的值（比如，在vector<int>中的int）。模板的参数应该是一个类型而不是值，所以我们使用typename关键字。紧跟着typename我们写了参数的名字T——整个和声明函数的参数很相似。当函数的调用者提供了一个模板的参数时，模板会把任何的引用当做这个参数T来处理，就如同它正是要处理的类型一样。同样的，就像使用函数参数一样来获取传递到函数中的值。

举个例子，如果调用者这么写：

```
triangleArea<double>( .5, .5 );
```

那么代码中出现 T 的任何地方，它都会被double代替。就仿佛我们写了triangleAreaDouble函数一样。我们写的代码在字面上就是个编译器用来创建专门用来处理double类型的模板。

换句话说，下面这行：

```
template <typename T>
```

可以这样理解：“接下来的函数（或者类）是个模板，在它内部，它将会使用字母T作为一个类型——就像int、double、char——或者某个其他类的名字。当有人需要使用这个模板时，必须为T提供一个特定的类型。通过把类型放在函数（或者类）名之前的尖括号（<>）中来实现。”

29.1.1　类型推断

在有些情况下，模板函数的调用者甚至都不需要显式地提供模板参数——编译器通常可以根据函数的参数来推断模板参数的值。举个例子，如果你这么写：

```
triangleArea( .5, .5);
```

编译器应当能够弄清楚T就应该是double。这是因为模板的参数T就是用来声明函数参数的。由于编译器知道了函数参数的类型，它就能推断出T应该是什么。

在任何时候，只要模板参数被用作函数的一个参数类型，类型推断就可以正常工作。

29.1.2　鸭子类型

有句话说如果它“看上去像只鸭子，走起来像只鸭子，说起话来也像只鸭子，那么它就是只鸭子”。神奇的是，这句话通常可以用来形容 C++ 模板相关的东西，原因如下：

当你传入一个模板参数时，编译器需要判断该模板参数对于模板来说是否合法。举个例子，在我们的compute_equation模板中，传入函数的值的类型必须能够支持数值运算符来进行加和乘的操作：

```
return x * y * 4 * z + y * z + x;
```

但是有些类型不能进行乘法操作。整型和双精度型，作为不同种类的数字，它们能够相乘。但如果是vector<int>怎么办呢？对一个vector进行乘法操作是很荒谬的——这没有任何意义而且vector类也不支持这个操作。

如果尝试传入三个 vector 到上面的compute_equation中，函数是无法编译通过的：

错误代码
```
int main ()
{
    vector<int> a, b, c;
    compute_equation( a, b, c );
}
```

实际上，编译器是很精准的，并且它还会告诉你哪些操作vector<int>是不支持的：

```
template_compile.cc: In function 'T compute_equation(T, T, T) [with T =
std::vector<int, std::allocator<int> >]':
template_compile.cc:13:    instantiated from here
template_compile.cc:5: error: no match for 'operator*' in 'y * z'
template_compile.cc:5: error: no match for 'operator*' in 'x * y'
```

这个错误信息很长，不过可以把它拆开来看。第一行告诉你哪个模板函数出了问题（compute_equation）；第二行告诉你哪一行代码在尝试使用该模板函数。这通常是你实际中要

到代码中去看的那一行。（顺便说一下，词语“instantiated from here”只是在说“这是你尝试使用模板的地方”。实例化是编程行话，指的是创建，在这个例子中，你尝试使用模板参数vector<int>去创建一个compute_equation的实现。）

下面的两行确切地告诉你为什么编译失败。在这个例子中，它说“no match for 'operator*' in 'x * y'”。这句话的意思是它无法弄清怎么去把x和y相乘（vector 没有定义 * 操作符）。由于两个变量都是 vector，你可以猜测这就意味着vector不支持乘法操作[①]。

vector换句话说，表现得不像一个数字——它没有“看上去像个数字，走起来像个数字，或者说起话来像个数字”。在使用一个模板函数的时候，编译器会去决定所给定的类型能不能在模板内部正常工作。它不关心别的，除了所给的类型是否支持需要调用的方法和操作。它只要“看上去像”一个可以正常工作的类型。

鸭子类型和多态函数的工作方式很不一样；一个多态函数接收一个指向接口类的指针并且只能够调用那个接口类里定义的方法。对于模板来说，模板参数不需要遵循预先定义的接口。只要是模板类型，那个类型的变量就能以函数所写的方式使用，函数会成功编译。换句话说，如果模板类型“看起来像个鸭子，走起路来像个鸭子，而且叫起来也像个鸭子”，我们的模板就会把它当做鸭子来处理。正常情况下，模板很少期待模板参数传入水生动物的描述，但是希望你现在明白为什么我们要说模板使用*鸭子类型*——重要之处就是传入的类型要能支持让模板正常工作的那些方法。

29.2　模板类

模板类通常是创建如vector和map这样的类的库函数作者用到的东西。但是日常编程也可以从创建更加通用的代码中获得好处。不要仅仅因为你会用模板就到处去使用模板，要注意寻找机会移除那些只有处理的类型不同，而其他都一样的类。模板类没有模板方法使用得普遍，但是知道怎么使用模板类很有好处——比如当你想要实现自己定制的数据结构时。

声明一个模板类和声明一个模板函数很像。

举个例子，可以创建一个小型的类来封装一个数组：[②]

```
template <typename T> class ArrayWrapper
{
private:
```

① 你可能会疑惑为什么编译器不在vector相加的时候报错。其实它会的，只是它还没能走到那一步。编译器看到了乘法操作存在问题并且在到达相加之前就退出了。

② 在编程中，封装这个术语使用的时候通常是，一个函数调用另一个函数来实现大部分的功能，但是外围的函数同时又去做一些不重要的想输出日志或者错误检查这样的额外工作。在这个例子中，主方法是用来实现外围方法的，而外围函数就可以说是封装了主方法。

```
    T *_p_mem;
};
```

就像写模板函数一样，一开始我们使用template关键字声明将要引入一个模板，然后在后面加上模板参数列表。这个例子中只有一个模板参数T。

我们在需要使用用户给定的类型的地方都用T——就像使用模板函数一样。

为模板类定义一个函数的时候，你必须也要使用模板语法。假设要在ArrayWrapper模板中添加一个构造函数：

```
template <typename T> class ArrayWrapper
{
public:
    ArrayWrapper (int size);
private:
    T *_p_mem;
};

// 现在，要在类的外部定义构造函数，作为开始，我们需要把函数标志为模板
template <typename T>
ArrayWrapper::ArrayWrapper (int size)
    : _p_mem( new T[ size ] )
{}
```

我们以相同的模板前奏为开始，再一次声明了模板参数。和之前唯一的不同就是类名包含了模板(ArrayWrapper<T>)，明确地表示了这是模板类的一部分，而不是一个叫做ArrayWrapper的非模板类的模板函数。

在这个方法实现中，我们可以用模板参数来代替调用者所要提供的类型，就像写模板函数时一样。和模板函数有所不同的是，模板类中的函数的调用者永远都不需要提供模板参数——参数是从初始的模板类型声明那里获取的。举个例子，当获取存储整型的 vector 的大小时，你不需要写vec.size<int>()或者vec<int>.size()，只要写vec.size()。

29.3　使用模板的一些小技巧

通常先为一个特定的类型写一个类，然后再用模板重写代码会更简单一些。举个例子，你可能声明一个使用整型的类，然后从这个声明想出一个通用的模板。这种方式不是必须的，如果你能很熟练地写模板的话就不需要用这个方式——但是在写自己的第一个模板时，它可以帮你把模板语法方面的问题从算法的问题中分离开来。

举个例子，来看一个起初只能处理整型的计算器类：

```
class Calc
{
```

```
public:
    Calc ();
    int multiply (int x, int y);
    int add (int x, int y);
};

Calc::Calc ()
{}

int Calc::multiply (int x, int y)
{
    return x * y;
}

int Calc::add (int x, int y)
{
    return x + y;
}
```

这个小巧的类能很好地处理整型。现在可以把它转成一个模板，那样就能创建非整型数据的计算器：

```
template <typename Type>
class Calc
{
public:
    Calc ();
    Type multiply (Type x, Type y);
    Type add (Type x, Type y);
};

template <typename Type> Calc<Type>::Calc ()
{}

template <typename Type> Type Calc<Type>::multiply (Type x, Type y)
{
    return x * y;
}

template <typename Type> Type Calc<Type>::add (Type x, Type y)
{
    return x + y;
}

int main ()
{
    // 展示如何声明
    Calc<int> c;
}
```

示例代码73：calc.cpp

这样的转换需要做几处修改：我们得声明有个模板类型叫做`Type`：

```
template <typename Type>
```

然后要在类以及函数定义之前加上这个模板声明：

```
template <typename Type> class Calc
```

```
template <typename Type> int Calc::multiply (int x, int y)
```

同样需要修改各个函数的定义来表明它是属于一个模板类：

```
template <typename Type> int Calc<Type>::multiply (int x, int y)
```

最后，要把所有int的地方都换成Type：

```
template <typename Type> Type Calc::multiply (Type x, Type y)
```

当你习惯了模板之后，把一个为特定类型而定义的类转换成一个很多类型都可以使用的模板类就是一个机械式地转换了[1]。随着时间的推移，你会熟练地使用模板的语法来从头写模板类而不需要任何的中间过渡代码。

模板和头文件

到目前为止我们看到的都是直接写在.cpp文件中的模板。如果想要把模板声明放到一个头文件中会怎样呢？问题在于使用模板函数（或者模板类）的代码对于每一次模板函数的调用（以及每次调用模板类的成员函数）都必须能够访问整个模板的定义。这和普通的函数工作原理很不一样，普通函数只要求调用者知道函数的声明。举个例子，假如你把Calc类放到了它自己的头文件中，你还得把构造函数的整个定义跟add方法也放到头文件中，而不是像平时一样把这些定义放到.cpp文件中。否则，任何使用Calc的尝试都会失败。

模板的这个让人遗憾的性质和模板被编译的方式有关；编辑器通常在第一次解析它们的时候会忽略这些模板。只有在你带着一个特定的具体类型来使用模板的时候（写Calc<int>的时候）编译器才会以这个特定的类型（在这个示例中就是int）来为模板生成代码。为了生成代码，大部分的编译器需要可以生成代码的模板。所以，你必须在每个使用模板的文件中包含所有的模板代码。再有就是，编译含有模板的文件时，你可能不会知道模板中的语法错误，直到第一次有人尝试使用这个模板。

当你创建一个模板类的时候，通常最简单的方式是单纯地把模板所有的定义都放到头文件中。使用一个不同于.h的扩展名来标明文件是个模板，这样会起到一定的帮助——比如，使用.hxx。

① 要小心避免过度范型化。例如，你有个循环计数器也是整型的，你不需要改变它的类型。

29.4　模板小结

模板让你可以创建通用的代码——可以服务于任何类型的代码，而不是被局限于，比如说，整型。模板被频繁地用来实现C++函数库（例如标准模板库）。你可能会发现不是经常需要写模板代码，但是要留意那些相同的结构而只是处理的数据类型不同的代码。举个例子，你可能会发现自己在写遍历不同类型的vector的代码，而且所执行的操作对每一个vector都是一样的。实际上，很多时候你需要模板，都是由于要处理另一个已经模板化的类型，例如STL的那些容器。

举个例子，你可能写了个函数来把vector中的数值加起来，还有一个函数把vector中的字符串拼接起来。这两个函数都有相同的基础结构，遍历一个vector以及使用+操作符，但是它们操作不同的数据类型。如果你见到这样的代码，遵循“不要重复自己”的原则。如果你写代码为两个不同的类型做同样的事，那就使用一个模板来代替写两种不同的实现吧。

诊断模板的错误信息

模板不好的一面就是大部分的编译器在你误用模板时都会给出难以理解的错误信息——哪怕模板不是你自己写的（比如说，这可能发生在使用STL的时候）。你可能因为一个失误导致被错误信息刷屏。模板错误信息很难读懂是因为它们把模板参数扩展成它们完整的类型了——甚至是你通常不会去使用的模板参数（因为它们被当成了默认参数）。

举个例子，看看这个貌似无辜的vector声明吧：

```
vector<int, int> vec;
```

这句声明有个小小的问题——它应该只有一个模板参数。不过你编译它的时候，会发现有超多的错误：

```
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In instantiation
of 'std::_Vector_base<int, int>':
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:159:
instantiated from 'std::vector<int, int>'
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:78: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:95: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:99: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In instantiation
```

```
of 'std::_Vector_base<int, int>::_Vector_impl':
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:123:
instantiated from 'std::_Vector_base<int, int>'
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:159:
instantiated from 'std::vector<int, int>'
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:82: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:86: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In instantiation
of 'std::vector<int, int>':
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:161: error: 'int'
is not a class, struct, or union type
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:193: error: no
members matching 'std::_Vector_base<int, int>::_M_get_Tp_allocator' in
'struct std::_Vector_base<int, int>'
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In destructor
'std::vector<_Tp, _Alloc>::~vector() [with _Tp = int, _Alloc = int]':
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:272: error:
'_M_get_Tp_allocator' was not declared in this scope
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In constructor
'std::_Vector_base<_Tp, _Alloc>::_Vector_base(const _Alloc&) [with _Tp = int,
_Alloc = int]':
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:203:
instantiated from 'std::vector<_Tp, _Alloc>::vector(const _Alloc&) [with _Tp
= int, _Alloc = int]'
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:107: error: no
matching function for call to 'std::_Vector_base<int,
int>::_Vector_impl::_Vector_impl(const int&)'
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:82: note:
candidates are: std::_Vector_base<int, int>::_Vector_impl::_Vector_impl(const
std::_Vector_base<int, int>::_Vector_impl&)
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In member
function 'void std::_Vector_base<_Tp, _Alloc>::_M_deallocate(_Tp*, size_t)
[with _Tp = int, _Alloc = int]':
/usr/lib/gcc/x86_64-redhatlinux/
```

```
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:119:
instantiated from 'std::_Vector_base<_Tp, _Alloc>::~_Vector_base() [with _Tp
= int, _Alloc = int]'
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:203:
instantiated from 'std::vector<_Tp, _Alloc>::vector(const _Alloc&) [with _Tp
= int, _Alloc = int]'
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhatlinux/
4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:133: error:
'struct std::_Vector_base<int, int>::_Vector_impl' has no member named
'deallocate'
```

到底出了什么状况？是谁在搞鬼，是谁故意给出这个错误信息的呢？问题是这样的：vector有第二个参数，它是个默认的模板参数——正常情况下编译器自动提供它。但是当你填入第二个int的时候，编译器会尝试使用int来作为第二个模板参数，然而这个参数又不能是个int。编译器实际上在错误列表开始的附近告诉你：

```
error: 'int' is not a class, struct, or union type
```

模板代码是在无法使用整型的情况下来尝试使用整型当模板参数。举个例子，如果有这样的代码：

```
template <typename T>
class Foo
{
    Foo ()
    {
        T x;
        x.val = 1;
    }
};
```

那么T就不能是个整型，因为x（T类型的）必须有个字段叫做val而整型根本没有任何字段，它们也就当然不存在叫做val的字段了。

如果这么写：

```
Foo<int> a;
```

代码就会编译失败。

这里又是鸭子类型（参见29.1.2节）——模板并不在意给它的确切类型，但是它在意所给的类型是否“适用于”代码。在这个例子中，一个整型无法支持“x.val”的语法，编译器没有让它通过。

vector模板对于它的第二个参数有个相似的约束——它需要一个能够比基础的整型支持更多功能的类型。所有的错误都是在抱怨，从很多方面来判断，int在这里是非法的类型！

面对着这么一大堆文字时，通常最好是从头开始查看错误信息并每次去尝试修复一个错误。我会略过别的信息直到我看到“error”的地方。

```
/usr/lib/gcc/x86_64-redhat-
linux/4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h: In instantiation of
'std::_Vector_base<int, int>':
/usr/lib/gcc/x86_64-redhat-
linux/4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:159:
instantiated from 'std::vector<int, int>'
template_err.cc:6:    instantiated from here
/usr/lib/gcc/x86_64-redhat-
linux/4.1.2/../../../../include/c++/4.1.2/bits/stl_vector.h:78: error: 'int' is not
a class, struct, or union type
```

好了，看起来好多了，对吧？只有几行了——很像我们之前关于鸭子类型的一节中看到的（参见29.1.2节）。这个我们能处理！

我们从头到尾看一下这个简单的错误信息。注意，第一行说“`In instantiation of 'std::_Vector_ base<int, int>'`”。模板的初始化表示“当尝试带着这一系列参数来编译模板的时候”。这个错误表示用那些参数来创建模板时出了问题（`Vector_base`是个用来实现vector的辅助类）。下面一行表示`Vector_bae`模板编译失败是因为一个创建`vector<int, int>`的尝试，并且它告诉你这个尝试来自`template_err.cc`的第6行；`template_err.cc`是我们自己的代码，所以现在知道导致错误的代码了。

找出有问题的那行代码通常是弄清楚出了什么错误的第一步。通常你只需看着自己的代码就可以辨别出什么地方出了错。如果第一眼看上去不是很明显，你可以继续追踪初始化的列表，直到找到真正的错误信息：`error: 'int' is not a class, struct, or union type`。这句话表明编译器希望你给的`int`能是一个类或者结构体，而不是语言自带的像`int`这样的类型。vector应当能够保存任意的类型，所以这暗示给vector的模板参数存在问题。到了这一步，你应当再次检查如何声明一个vector，然后会看到其实只需要一个模板参数。

既然对第一个问题有了诊断结果，是时候修正它并且重新编译了。正常情况下你要一次处理至少几个编译错误，但是，对于模板而言第一个错误通常会导致其他所有的错误。最好一次修正一个问题，这样它们就不会再让你头疼了，然后你再接着修正其他的。

在这个例子里，超过一页的错误信息中，每个错误都是由于添加了第二个`int`模板参数。

29.5 问答题

(1) 什么时候应该使用模板？

A. 想要节省时间的时候

B. 想要代码运行得更快的时候
C. 需要为不同类型多次写相同代码的时候
D. 需要确保之后可以重用代码的时候

(2) 你什么时候需要为模板参数提供一个类型？

A. 总是需要
B. 只有在声明一个模板类的实例的时候
C. 只有类型无法推断出来的时候
D. 对于模板函数，只有在类型无法推断的时候，对于模板类，一直都需要

(3) 编译器如何辨别一个模板参数可以用于一个给定的模板？

A. 它会实现一个特定的C++接口
B. 声明模板的时候你必须指定约束条件
C. 它会尝试使用模板参数；如果参数类型支持所有需要的操作，编译器就会接受它
D. 在声明模板的时候你必须列出所有合法的模板类型

(4) 把模板类放在头文件中和把一个常规的类放在头文件中有什么不同？

A. 没什么区别
B. 常规类不能在头文件中定义它的任何方法
C. 模板类必须把所有的方法在头文件中定义
D. 模板类不需要有对应的`.cpp`文件，但是常规的类需要有

(5) 什么时候应该把函数写成模板函数？

A. 一开始的时候——你永远不会知道什么时候需要对不同的类型使用相同的逻辑，所以你应当总是写模板方法
B. 只有在你无法把所给的类型转换成函数当前需要的类型的时候
C. 当你写了几乎一样的逻辑，但是处理的是一个与第一个函数所使用的类型有着相似特性的不同的类型的时候
D. 当两个函数做“几乎是”相同的事，而且你可以通过几个额外的 `Boolean` 参数就能把逻辑修改过来的时候

29

(6) 你什么时候会知道所写模板存在的大部分错误？

A. 在你编译模板的时候
B. 在链接的阶段
C. 在你运行代码的时候
D. 在你第一次编译初始化模板的代码时

29.6 实践题

(1) 写个函数接收一个vector并且把vector中所有值求和，不管vector中存储的是什么类型的数值。

(2) 修改在第24章中实现的vector替代类，把它变成一个模板让它可以存储任意的类型。

(3) 写个搜索方法，接收一个任意类型的vector以及一个任意类型的值，如果值在vector中就返回`true`，否则返回`false`。

(4) 实现一个排序函数，接收一个任意类型的vector然后根据自然顺序给vector中的值排序（你通过<或>得出的顺序）。

Part 4 第四部分

其　　他

你已经了解了编写有趣的、大规模的程序所需要的工具。还有几个主题，虽然有用，但不适合在本书中叙述；这些主题包括从命令行得到参数以及执行良好的输入和输出格式化。这些主题相对于算法逻辑而言，与用户界面程序的相关性更大，但它们同样重要。只有与用户交流，程序才会变得非常有趣！

你可以按任意顺序阅读这部分的主题，这取决于想要完成的任务。你甚至应该在完成本书其他部分之前，先读这部分的一些内容，尤其是你在课堂学习的部分涉及这些主题时。

第 30 章

使用iomanip格式化输出

“讨厌”的终端用户通常要求你创建干净整齐的格式化输出。（下一件事你知道的，就是让程序能够正常工作！）在C++里面，你可以使用iomanip头文件里的函数配合cout创建漂亮的格式化输出。

30.1 处理空间问题

最常见的格式问题是间距处理不当。良好格式化的输出使用的间隔看起来刚刚好。列的文本不会太长或太短，一切都对齐得当。所以，让我们看看如何做到吧！

30.1.1 使用setw设置字段宽度

setw函数使你能够通过插入操作符设置下一个输出的最小宽度。如果下一个输出小于最小宽度，就使用空格填充输出。如果输出比最小宽度长，什么也不做——重要的是，输出是不会被截断的。

实际使用setw有点奇怪——你调用该函数并把数值传递到cout：

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << setw( 10 ) << "ten" << "four" << "four";
}
```

示例代码74：setw.cpp

以上程序的输出为：

```
tenfourfour
```

如果你调用setw但是没有把它传递给cout，无论如何都没有任何效果。正如你从示例程序

中看到的，对于setw的调用仅仅影响紧接着的下一个输出。

你会发现，在默认情况下，字符串是右对齐的（填充是放在字符串的左边）。换而言之，字符串以填充字符作为前缀。你可以通过传递对齐的方向left或者right到cout中，来设置想要的输出对齐方式。这个示例程序将文本左对齐，而不是右对齐，使输出更加可读。

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << setw( 10 ) << left << "ten" << "four" << "four";
}
```

示例代码75：setw_left.cpp

上面的输出应该像这样：

```
ten       fourfour
```

结果是左对齐的。

setw允许你在运行时决定输出的列的宽度。例如，为了显示几列的数据，你可以找出每一列最宽字符串，填补该列的每个条目，这样每一个条目都比该列最长的元素稍宽一点。

30.1.2 改变填充字符

有些时候，你可能不希望使用空格填充。你可以调用setfill改变填充字符。setfill的工作模式类似setw，你也可以直接传递给cout。

如果我们还是使用原先填充的例子，但添加一个破折号的setfill：

```
cout << setfill( '-' ) << setw( 10 ) << "ten" << "four" << "four";
```

它看起来就会像这样：

```
-------tenfourfour
```

30.1.3 永久改变设置

你也可以使用cout的fill成员函数来全局改变填充字符。例如，这个代码：

```
cout.fill( '-' );

cout << setw( 10 ) << "A" << setw( 10 ) << "B" << setw( 10 ) << "C" << endl;
```

将输出：

```
---------A---------B---------C
```

fill方法返回之前的填充字符，这样之后你就可以恢复它。如果你意欲避免多次setfill的调用，那么它的返回是可以利用的。例如：

```
const char last_fill = cout.fill( '-' );

cout << setw( 10 ) << "A" << setw( 10 ) << "B" << setw( 10 ) << "C" << endl;

cout.fill( last_fill );

cout << setw( 10 ) << "D" << endl;
```

现在最后一行则输出为：

```
         D
```

你可以通过调用cout的setf成员函数永久设置填充文本的对齐。你可以把标志位传递到setf来设定向左或者向右的功能，使标志位ios_base::left或者ios_base::right[①]。

```
cout.setf( ios_base::left );
```

使用fill，这个访问就可以返回上一个值，以便你之后可能想要恢复它。

试着把上面的对setf的调用添加到之前的例子来看一下格式的不同之处。

30.2　把你的 iomanip 知识汇总到一起

让我们把上面的一些方法放在一起，并且编写代码使姓和名成为两列，保证这两列对齐良好，就像这样：

```
Joe    Smith
Tonya  Malligans
Jerome Noboggins
Mary   Suzie-Purple
```

我们需要正确的设置列宽，仅仅比每一列最大的元素大一点点。我们可以遍历代码，发现最大的长度，然后使用setw设置最大长度（选择性地增加一些填充）来展示这些名字。来看看实现它的代码：

```
#include <iostream>
#include <vector>
#include <iomanip>
```

① setf表示set flag，也就是设置标志位。

```
using namespace std;

struct Person
{
    Person (
        const string& firstname,
        const string& lastname
    )
    : _firstname( firstname )
    , _lastname( lastname )
    {}

    string _firstname;
    string _lastname;
};

int main ()
{
    vector<Person> people;

    people.push_back( Person( "Joe", "Smith" ) );
    people.push_back( Person( "Tonya", "Malligans" ) );
    people.push_back( Person( "Jerome", "Noboggins" ) );
    people.push_back( Person( "Mary", "Suzie-Purple" ) );

    int firstname_max_width = 0;
    int lastname_max_width = 0;

    // 获取最大宽度

    for ( vector<Person>::iterator iter = people.begin();
        iter != people.end();
        ++iter )
    {
        if ( iter->_firstname.length() > firstname_max_width )
        {
            firstname_max_width = iter->_firstname.length();
        }
        if ( iter->_lastname.length() > lastname_max_width )
        {
            lastname_max_width = iter->_lastname.length();
        }
    }

    // 输出vector中的元素
    for ( vector<Person>::iterator iter = people.begin();
        iter != people.end();
        ++iter )
    {
        cout << setw( firstname_max_width ) << left << iter->_firstname;
        cout << " ";
        cout << setw( lastname_max_width ) << left << iter->_lastname;
        cout << endl;
    }
}
```

示例代码76：column_alignment.cpp

30.2.1 输出数字

创建漂亮的输出有时需要正确格式化数字；当输出一个十六进制值，加一个前缀0x来显示数值的进制就很好。如果你把小数点后面0的个数与应用程序相适应（例如设置为2，如果在处理与金钱有关的程序）那就更好了。

30.2.2 使用**setprecision**来设置数值输出的精度

setprecision函数用来在输出一个数字时设置数字位数的最大值。就像setw，setprecision的返回值应该插入到流中。实际上，它的使用在各方面都与setw相似。设定数字2.718 28输出时总共有3位：

```
std::cout << setprecision( 3 ) << 2.71828;
```

调用setprecision将会适当地近似化输出——因此，这里的输出是2.72，而不是直接缩短变成2.71。另一方面，如果要输出的是2.713，那结果将变成2.71。

不像其他插入到流中的命令，setprecision将改变精确度直到下一次它被传递到一个给定的流。所以像这样改变上面的例子：

```
cout << setprecision( 3 ) << 2.71828 << endl;
cout << 1.412 << endl;
```

将输出：

```
2.72
1.41
```

如果输出的数值在小数点前面比setprecision所提供的精度有更多的数字，你可能会好奇结果会变成什么。答案取决于是否输出一个浮点数或整数。整数全部输出，浮点数按照要求位数的数字以科学记数法输出：

```
cout << setprecision( 2 ) << 1234.0 << endl;
```

文本中的结果为：

```
1.2e3
```

顺便提一下，这个e3指的是10^3。

然而：

```
cout << setprecision( 2 ) << 1234 << endl;
```

文本中的结果则为：

```
1234
```

30.2.3 如何处理货币

到现在为止你可能已经注意到，还没有一个好的办法可以输出代表货币的数值，通常你总是想在小数点后有两位数字，但是又不想要任何的近似。

简短的回答是，可能无论如何你都不应该将货币存储为double！原因是double不完全精确，所以可能会引入小的近似误差，不是在这儿就是在那儿去除掉几分钱。对于大部分应用程序来说，一个更好的存储货币的方式是把总的美分数存储为整数。当你想要显示该数值时，为了更高的精确度，可以除以100来得到美元数，然后取模来得到美分数，同时分开显示每一个值。

```
int cents = 1001; // $10.01
cout << cents / 100 << "." << cents % 100;
```

当然，创建一个标准的帮助函数来为你进行这种计算，并且创建一个类来存储货币，隐藏使用数值格式的具体细节，这么做是有意义的。

30.2.4 按不同的进制输出

编程的时候，你经常想用八进制或十六进制显示数字。你可以使用setbase函数来完成。当插入一个流时，setbase设置进制为8、10或者16。举个例子，

```
cout << "0x" << setbase(16) << 32 << endl;
```

将输出为：

```
0x20
```

这是32的十六进制写法。注意你可以分别使用dec、oct和hex，分别作为setbase(10)、setbase(8)和setbase(16)的缩写插入到流中。

尽管上面的代码明确的输出0x，你可以使用setiosflags指示cout应该自动显示进制。如果你把setiosflags(ios_base::showbase)的结果传递到cout，然后十进制将正常显示，十六进制数字将使用0x前缀，八进制数字将前缀0。

```
cout << setiosflags( ios_base::showbase ) << setbase( 16 ) << 32 << endl;
```

将输出：

```
0x20
```

类似setprecision，由setiosflags产生的变化是永久性的。你可以使用noshowbase作为参数来禁用前缀。

有这些工具在手，你应该能够创建更加合意的输出。

第 31 章

异常和错误报告

构建更大的程序时，你需要一个简洁的方式来处理函数的错误报告。报告错误有两个经典方式：使用错误代码和使用异常。使用错误代码不需要任何新的语言特性，但是这意味着每个函数（可能会失败的）要返回一个错误代码（或成功的代码），表示函数运行的结果。这种技术的优点是相对容易理解：

```
int failableFunction ();

const int result = failableFunction();

if ( result != 0 )
{
    cout << "Function call failed: " << result;
}
```

另一方面，这个错误代码的处理技术具有要求每个函数返回一个错误代码的缺点，即使你想要从函数中获得另一个的值。为了使函数返回一个计算后的值，你需要使用一个引用或指针参数：

```
int failableFunction (int& out_val);

int res_val;

const int result = failableFunction( res_val );

if ( result != 0 )
{
    cout << "Function call failed: " << result;
}
else
{
    // 利用 res_val 做点儿什么
}
```

虽然这个方法可行，但是代码再也不能显示你所期待的自然的流程。

另一方面，异常是一个全新的语言特征。异常作用的方式是当一个函数想报告一个错误时，它立即停止执行，并且抛出异常。当一个异常被抛出时，程序搜索一个异常处理器，它将处理这

个异常。

一种理解异常的方式是假设函数立即返回，而且没有返回值。此外，不同于返回到函数的调用者，程序的执行返回到可以真正的处理异常的地方。如果没有可以返回的地方，那么程序将会由于一个未处理的异常而崩溃。否则，它将返回到处理异常的地方，程序将从该处继续执行。这允许您编写代码，在失败“返回”到的地方，立即处理所有这些问题。

为了指定一个执行失败的函数应该返回的地方，你可以使用一个try-catch区块：

```
try
{
    // 可能失败并抛出异常的代码
}
catch ( ... )
{
    // 处理异常的地方（也就是，函数返回到的地方）
}
```

任何在try区块的函数都可以抛出一个异常，该异常随后会在catch区块得到处理。可以有许多种异常，每一个类别都不一样，这样使你可以写多个catch区块，每个catch区块处理不同种类的失败。如果你使用catch...，正如上面的代码那样，那么任何一个没有被别的更具体的catch区块捕获到的异常都会被那个catch区块处理。你可以把...想成捕获所有的。第一个能够处理某异常的catch区块将处理该异常，这样如果你想有一个catch all，你应该把它放在最后，在所有catch区块的后面：

```
try
{
    // 可能失败并抛出异常的代码
}
catch ( const FileNotFoundException& e )
{
    // 处理由于找不到文件导致的异常
}
catch ( const HardDriveFullException& e )
{
    // 处理硬盘空间不够导致的异常
}
catch ( ... )
{
    // 处理其他异常的地方（也就是，函数返回到的地方）
}
```

异常时释放资源

如果调用的函数抛出了一个异常，不是非要去捕获它——这个异常将从你的函数传递出来，它可能会在一个更上层的函数中找到一个catch区块。只要不需要对异常做出任何反应，这样是

完全正确的。实际上，你通常是不需要做任何事情，因为当函数由于一个异常而退出时，所有的本地对象的析构函数将被调用。例如：

```
int callFailableFunction ()
{
    const string val( "abc" );
    // 调用抛出异常的代码
    failableFunction();
}

int main ()
{
    try
    {
        callFailableFunction();
    }
    catch ( ... )
    {
        // 处理异常
    }
}
```

这里，如果failableFunction抛出一个异常，那么构建于callFailableFunction的val字符串将会被销毁，所有分配用来存储字符串的资源将会被清理。这一特性称为栈展开（stack unwinding）——通过调用在该结构内每个对象的析构函数，各个没有捕捉到异常的堆栈结构都被清除，或者展开。记住，即使你没有显式地写一个析构函数，这些对象也有默认的析构函数做一些清理工作。

手动在catch区块中清理资源

有时抛出一个异常时，你确实需要手动的清理一些资源。大部分情况下，你应该尝试写一个保护对象来清理该资源，但是如果没有做这个选择，你可以总是捕捉异常，清理资源，然后重新抛出异常。例如：

```
int callFailableFunction ()
{
    const int* val = new int;
    // 调用抛出异常的代码
    try
    {
        failableFunction();
    }
    catch ( ... )
    {
        delete val;
        // 注意这里使用throw;来重新抛出异常
        throw;
    }
```

```
    delete val; // 注意，我们也必须在这里放一个delete
                // 如果没有异常catch区块就不会执行
                // 保证代码总是会执行到的唯一方法，就是把它
                // 放在一个局部变量的析构函数中
}

int main ()
{
    try
    {
        callFailableFunction();
    }
    catch ( ... )
    {
        // 失败处理
    }
}
```

抛出异常

目前为止，你已经看到了很多有关如何捕捉或者处理异常的例子——但是如何创建和抛出异常呢？创建一个异常类没什么特别的——它只是一个普通的类。你把任何觉得重要的字段都放入该异常中，并且提供访问方法来读取异常相关的信息。一个典型的异常将有类似这样的接口：

```
class Exception
{
public:
    virtual ~Exception () = 0;
    virtual int getErrorCode () = 0;
    virtual string getErrorReport () = 0;
};
```

然后各个具体类型的错误都去继承Exception并且实现这些虚方法：

```
class FileNotFoundException : public Exception
{
public:

    FileNotFoundException (int err_code, const string& details)
        : _err_code( err_code )
        , _details( details )
    {}

    virtual ~FileNotFoundException ()
    {}

    virtual int getErrorCode ()
    {
        return _err_code;
    }

    virtual string getErrorReport ()
```

```
    {
        return _details;
    }

private:

    int _err_code;
    string _details;
};
```

接着你可以抛出异常，就好像是在构造这个类的一个实例：

```
throw FileNotFoundException( 1, "File not found" );
```

从通用的基类继承所有异常的一个优势是这些异常可以被父类捕捉到。例如，可以写：

```
catch ( const Exception& e )
{
}
```

接着，它将捕捉继承自Exception类的任何异常。使用一个精心定义的异常层次结构可以让你在单个catch区块中处理各种各样的错误。例如，所有输入和输出错误可能会继承叫做IOException的一个类，从而允许所有IO异常作为一个单独的单元来处理——在具体的情况下，不同的子类需要不同的处理，这段代码仍然可以捕捉IOException的特定子类。

在标准库中构建的原生的C++标准异常父类是std::exception。你不需要以它作为父类来定义自己的异常层次结构，但是如果要使用标准库，使用std::exception作为一个公共基础类就很有意义，这样可以使用它来捕捉程序中抛出的所有异常——包括来自标准库的和自己代码的。

异常抛出说明

好了，现在当到遇一个错误可以抛出一个异常，并且如果知道一个函数将会失败你可以捕获到一个异常。好吧，但是怎么知道一个函数是否会抛出异常呢？在C++中，你可以通过throw spec来指定你期望你的函数可能抛出的异常。throw spec是一个出现在函数声明和函数定义末尾的异常的列表，它可能是空的。

在头文件中：

```
void canFail () throw (FileNotFoundException);
void cannotFail () throw ();
```

在cpp文件中：

```
void canFail () throw (FileNotFoundException)
{
    throw FileNotFoundException();
}
```

```
void cannotFail () throw ()
{
}
```

异常说明的问题是，它们在编译时是不检查的；它们只有在运行时才被检查。更糟的是，如果一个函数抛出一个预期之外的异常，程序可能会立即终止。这意味着你不能真正的依赖于异常说明的准确性，但倒是完全可以期待它们引发你的程序崩溃。一些工具，例如PC-Lint（http://www.gimpel.com/html/pcl.htm），提供编译时的异常检查，并且缓解了人们使用异常说明时遇到的许多问题。在新的C++标准中，也就是C++11，完整的异常说明已经被弃用，这意味着它们在未来不太可能继续作为语言的一部分了[①]。

这导致的最终结果是，你必须依靠函数的作者来正确地注明函数可能抛出的异常，如果写的一个函数会抛出一个异常，需要注明它会抛出一个异常。

异常的好处

异常有两个主要的好处，一是通过把错误都放进一个单一的catch区块，而不是必须去做很多的返回码检查，这样简化了错误处理逻辑；二是通过给出更多信息而不只是一个错误码来改善错误报告。

第一个好处，允许错误在单个catch区块中处理，可以把这样的代码：

```
if ( funCall1() == ERROR )
{
    // 处理错误
}
if ( funCall2() == ERROR )
{
    // 处理错误
}
if ( funCall3() == ERROR )
{
    // 处理错误
}
```

转变成：

```
try
{
    funCall1();
    funCall2();
    funCall3();
}
catch ( const Exception& e )
```


① 异常说明保留着一个功能，就是说明一个函数肯定不会抛出异常，这有时能够改善性能。

```
{
    // 处理错误
}
```

所有的错误处理代码在一个地方，并且主线使用示例遵循起来非常简单。

第二个好处，允许错误报告附加信息也非常有用。使用一个错误代码，你也只能得到该错误代码。使用异常时，每个错误可以提供相关的附加信息。例如一个FileNotFoundException可以包含文件的名字。

异常的错误使用

尽管异常是报告错误的神奇工具，但是由于它们具有从函数中立即返回到堆栈上的一个更早的调用者的能力，它们也可能被滥用。总的来说，你不应该用异常来处理预期的，非错误的状况。例如，理论上来讲你可以使用一个异常来报告一个函数执行的结果而不是返回一个值。但是这样做比返回值更慢（需要一些运行时间成本来处理一个抛出的异常）并且会令人困惑。正如你在上面见到的，使用异常报告错误简化了函数的主线逻辑。如果你开始使用异常作为主线逻辑的一部分，那么就失去了那个简化作用。

来看一下你如何用解析器代码的一个片段作为例子，把使用异常的主线使用案例的代码，在不使用异常的情况下重新编写。解析器是一段代码，它读入一个定义良好的语言——例如HTML——并解释其结构。通常一个解析器会有解析程序结构中的单个元素的函数。例如，在HTML中，就可能有函数来解析链接和表格。

一种编写解析器的方式是使每个函数，即parseLink和parseTable报告它们是否能够解析下一段文本，如果不能则使用一个异常：

```
try
{
    parseLink();
    return;
}
catch ( const ParseException& e )
{
    // 不是链接，试试下一个类型
}
try
{
    parseTable();
    return;
}
catch ( const ParseException& e )
{
    // 不是表格，试试下一个类型
}
```

这里的问题是，如果第二块文本不是一个链接或一个表，这并不是一个错误，而是正常情况。最好这样写解析器：

```
if ( expectLink() )
{
    parseLink();
}
else if ( expectTable() )
{
    parseTable();
}
```

由于HTML通常使用几个字符来表明页面上的下一个元素是什么，你可以轻松地写个方法检查文档的下一个部分是否为链接或表格，然后就可以使用简单的if语句，而不是复杂的异常。

异常的总结

异常是报告错误的一种简洁的方式，无需因为特定错误处理逻辑把代码弄得乱七八糟。归功于清理对象的栈展开和析构函数，异常允许你的大部分代码遵循算法的主线逻辑而不要检查错误代码。

抛出异常确实有一些性能影响，所以你应当在错误发生时使用异常，而不是作为算法控制流程的一部分。例如，如果读入非法的字符，解析器可能会抛出异常；在遇到属于正常文件格式的字符的情况下，它不应该抛出异常。这使得哪种情况才是真正的错误变得很清楚。只有在罕见的，有一个真正的问题出现时才运行异常处理，这也保证了代码的最佳性能。这些情况几乎总是会导致算法的终止，所以如果它们比平时运行得慢也没关系。

在许多真实的程序中，错误处理在产品开发所需的时间上占据着主要的比重，所以当看过了我们在这本书中提到的基础内容后，你会看到和听到更多关于异常的内容。

第 32 章

最后的话

32

现在你已经学到了关于C++的很多知识，但是学习之旅并未结束。说实在的，你真的还处在终生学习编程的开始阶段。现在你掌握了可以编写很多有趣的、复杂的程序的工具，下一步就是放手去做：构建复杂的系统以及练习实现算法和数据结构。在你所使用的语言之外还有很多关于编程的话题：如何设计程序、如何设计算法、如何设计用户界面、应该使用哪些类库、如何组织程序员团队，甚至还有如何确定第一步要构建什么。换句话说，有许多软件项目需要去做。本书已经涉及这些领域中的一部分，但它们本身都是一些完整的主题，不可小觑。

如同学习任何一门人类语言，除了语言的基础语法和句式还有太多东西要学习。你无法从会说英语直接跳到可以写一部伟大的小说。同样，你也无法从会写C++直接到可以创建一个操作系统。但重要的是，现在你已经具备了学习更多概念和思想的基础，这些概念和思想是进入下一个境界所需要的。下面是关于接下来应当做些什么的建议。[①]

(1) 读一些关于软件工程和算法设计方面的书。像Jon Bentley的《编程珠玑》（*Programming Pearls*）对一些非编程语言层面的编程知识进行了生动有趣的介绍，包括基本的算法分析、设计和评估。

(2) 多写程序。从模仿已有的软件开始，写一些已有工具的克隆版本，学习那些开发这类工具所需要用到的类库。接着参与到开发中——找一个实习职位或者参加一个开源项目。代码量越大，越可能写出糟糕的代码，但只有通过写糟糕的代码才能最终学会写优秀的代码。

(3) 阅读其他方面的知识——不仅仅是编程。学习软件测试、项目管理、产品管理、市场营销；最终，你对整个软件开发过程理解得越多，距离成为一名全面的开发者、架构师或总经理就越近。

(4) 找到别的程序员，和他们一起工作，向他们学习。这是在大学里上一门课程或者找个实习工作的一个好处。

(5) 找一个过来人做导师。书页上的文字无法回答作者没有想过的问题；找到一个像你一样的人可以帮你跳过很多障碍。要足够恭敬有礼，但是不要怕问问题，也不要怕暴露你所未知的东西。迷惑不解的时候是个学习的好机会。

---

① 我的一些建议受到了Peter Norvig的“Teach Yourself Programming in Ten Years”（http://norvig.com/21-days.html）一文的启发。

(6) 享受编程。如果你没感受到乐趣，那么可能不想把它当做职业生涯全职去做。保持乐趣，不要去做那些让你不想写程序的无聊的事。

现在本书的阅读行将结束，但你的生涯才刚刚开始。祝你好运！

第 2 章问答题答案

(1) 程序正确执行后，会返回给操作系统什么值？

A. -1 B. 1 **C. 0** D. 程序不返回值

(2) 所有C++必须包含的函数是？

A. start() B. system() **C. main()** D. program()

(3) 什么符号用于表示代码段的开始和结尾？

A. { } B. -> 和 <-
C. BEGIN 和 END D. (和)

(4) 大部分C++程序以什么符号结尾？

A. . **B. ;** C. : D. '

(5) 下面哪个是正确的注释符号？

A. */ Comments * B. ** Comment **
C. /* Comment */ D. { Comment }

(6) 使用cout需要包含哪个头文件？

A. stream B. 不需要包含（默认包含）
C. iostream D. using namespace std;

第 3 章问答题答案

(1) 什么变量类型可以存储数值3.1315?

A. int B. char **C. double** D. string

(2) 下面哪个是比较两个变量的操作符？

A. := B. = C. equal **D. ==**

(3) 如何获取string数据类型?

A. 语言中包含，无需任何操作
B. 因为字符串用在输入输出上, 你需要引用iostream头文件
C. 引用string头文件
D. C++不支持

(4) 下面哪个变量类型不正确?

A. double　　**B. real**　　C. int　　D. char

(5) 怎么读取用户的一整行输入?

A. 使用cin >>　　B.使用readline　　**C. 使用getline**　　D. 很困难

(6) C++中，cout << 1234/2000会输出什么结果?

A. 0
B. 0.617
C. 大约0.617，不过结果不能精确地存储在浮点数中
D. 要看等式两边的类型

(7) 为什么C++在有整数类型的情况下还需要char类型?

A. 因为字符和整数是两种完全不同的类型，一个是数字，一个是字母
B. 为了向下兼容C
C. 字符比数字更加容易读入和输出，尽管字符实际上被存储为数字
D. 对国际支持，处理像汉语和日语这种包含很多字符的语言

第 4 章问答题答案

(1) 下面哪个是true?

A. 1　　B. 66　　C. 0.1　　D. -1
E. 以上全部

(2) 下面哪个是逻辑与的操作符?

A. &　　**B. &&**　　C. |　　D. |&

(3) 表达式!(true && ! (false || true))的结果是?

A. true　　B. false

(4) 下面哪个是if语句的正确语法?

A. if expression　　B. if { expression
C. if (expression)　　D. expression

第 5 章问答题答案

(1) 代码int x; for(x=0; x<10; x++) {}中，x最终的值是?

A. 10　　B. 9　　C. 0　　D. 1

（如果这让你困惑，想想如果在for循环后加一个cout语句会发生什么。）

(2) while(x<100)之后的代码何时会执行?

A. 当x小于100　　B. 当x大于100　　C. 当x等于100　　D. 当它愿意的时候

(3) 哪个不是循环结构?

A. for　　B. do-while　　C. while　　**D. repeat until**

(4) do-while能保证循环几次?

A. 0　　B. 无限次　　**C. 1**　　D. 不定

第 6 章问答题答案

(1) 哪个不是正确的原型?

A. int funct(char x, char y);　　**B. double funct(char x)**
C. void funct();　　D. char x();

（注意丢失的分号）

(2) 函数原型int func(char x, double v, float t);的返回值类型是什么?

A. char　　**B. int**　　C. float　　D. double

(3) 下面哪个函数调用是有效的（假设函数存在）?

A. funct;　　B. funct x, y;　　**C. funct();**　　D. **int funct();**

(4) 下面哪个是完整的函数?

A. int funct();

B. `int funct(int x) {return x=x+1;}`

C. `void funct(int) {cout << "Hello"}`

D. `void funct(x) {cout << "Hello";}`

第 8 章问答题答案

(1) 紧接着case语句的是什么？

A. : B. ; C. - D. 换行

(2) 避免从一个case执行到另一个case需要什么？

A. end; **B. break;** C. Stop; D. 分号

(3) 哪个关键字用来处理未知情况？

A. all B. contingency **C. default** D. other

(4) 下面代码的运行结果是？

```
int x = 0;
switch( x )
{
    case 1: cout << "One";
    case 0: cout << "Zero";
    case 2: cout << "Hello World";
}
```

A. One B. Zero C. Hello World **D. ZeroHello World**

第 9 章问答题答案

(1) 在rand之前不调用srand会发生什么？

A. rand失败

B. rand一直返回0

C. rand会在每次程序运行时返回同样的数列

D. 什么都不发生

(2) 为什么要用当前时间作为srand的种子？

A. 确保程序运行一致

B. 每次程序运行时产生新的随机值

C. 确保计算机产生真随机数

D. 这样做是为了方便你在对同一个操作需要多次设置种子时，只需调用一次`srand`

(3) `rand`返回值的范围？

A. 想多少就多少　　B. 0 ~ 1000

C. 0 ~ `RAND_MAX`　　D. 1 ~ `RAND_MAX`

(4) 表达式`11 % 3`的结果是？

A. `33`　　B. `3`　　C. `8`　　**D. `2`**

(5) 什么时候应该使用`srand`？

A. 每次需要随机值时

B. 永远不需要，这只是Windows的一个封装

C. 一次，在程序的开头

D. 偶尔，在`while`循环内使用`rand`之后增加随机性

第 10 章问答题答案

(1) 下面的声明数组中哪个正确的？

A. `int anarray[ 10 ];`　　B. `int anarray;`

C. `anarray{ 10 };`　　D. `array anarray[ 10 ];`

(2) 有29个元素的数组的最后一个元素的索引是什么？

A. `29`　　**B. `28`**　　C. `0`　　D. 程序定义的索引

(3) 下面哪个是多维数组？

A. `array anarray[ 20 ][ 20 ];`　　**B. `int anarray[ 20 ][ 20 ];`**

C. `int array[ 20, 20 ];`　　D. `char array[ 20 ];`

(4) 一个有100个元素的数组`foo`，下面哪个可以正确访问第7个元素？

A. `foo[ 6 ];`　　B. `foo[ 7 ];`　　C. `foo( 7 );`　　D. `foo;`

(5) 下面哪个函数可以接受二维数组？

A. `int func ( int x[][] );`　　B. `int func ( int x[ 10 ][] );`

C. `int func ( int x[] );`　　**D. `int func ( int x[][ 10 ] );`**

第 11 章问答题答案

(1) 下列哪个选项访问了结构体b里的变量？

A. b->var;　　B. b.var;　　C. b-var;　　D. b>var;

(2) 下列哪一项正确定义了一个结构体？

A. struct {int a;}　　B. struct a_struct {int a};
C. struct a_struct int a;　　**D. struct a_struct {int a;};**

(3) 下列的哪个选项正确地声明了类型名为foo，变量名为my_foo的结构体变量？

A. my_foo as struct foo;　　**B. foo my_foo;**
C. my_foo;　　D. int my_foo;

(4) 以下代码的最终输出结果是多少？

```
#include <iostream>

using namespace std;

struct MyStruct
{
    int x;
};

void updateStruct (MyStruct my_struct)
{
    my_struct.x = 10;
}

int main ()
{
    MyStruct my_struct;
    my_struct.x = 5;
    updateStruct( my_struct );
    cout << my_struct.x << '\n';
}
```

A. 5　　B. 10　　C. 此代码无法编译

第 12 章问答题答案

(1) 以下哪项不是使用指针的好理由？

A. 你想要允许函数修改传递给它的参数
B. 你想要避免复制一个占用了很大的内存的变量，以节省空间

C. 你希望能够从操作系统获得更多的内存

D. 你希望能够更快速地访问变量

(2) 指针中存储的是什么？

A. 另一个变量的名称

B. 一个整数值

C. 另一个变量在内存中的地址

D. 一个内存地址，但不一定是另一个变量

(3) 程序执行过程中，从哪里可以获取到更多的内存？

A. 不能得到任何更多的内存

B. 栈

C. 未分配内存区域

D. 通过声明另一个变量

(4) 使用指针时，可能会遇到什么错误？

A. 访问了本不能用到的内存，导致崩溃

B. 访问错误的内存地址，导致数据污染

C. 忘了将内存释放回操作系统，导致程序耗光内存

D. 以上皆可

(5) 函数中声明的普通变量，其内存来自哪里？

A. 未分配存储区域

B. 栈

C. 普通变量不使用内存

D.来自该程序的二进制文件本身（这就是为什么EXE文件会这么大！）

(6) 分配到内存后，需要做些什么？

A. 什么都不用做，它永远是你的

B. 使用完后要释放回操作系统

C. 将所指向的值置为0

D. 将值0存入指针中

第 13 章问答题答案

(1) 下面哪个选项正确地声明了指针？

A. int x;　　B. int &x;　　C. ptr x;　　**D. int *x;**

(2) 下面哪一项给出了整数变量的内存地址?

A. *a;　　B. a;　　**C. &a;**　　D. address(a);

(3) 下面哪一项给出了指针p_a所指向变量的内存地址?

A. p_a;　　B. *p_a;　　**C. &p_a;**　　D. address(p_a);

(4) 下面哪一项给出了指针p_a所指向地址中存储的值?

A. p_a;　　B. val(p_a);　　**C. *p_a;**　　D. &p_a;

(5) 下列哪一项正确声明了一个引用?

A. int *p_int;　　B. int &my_ref;
C. int &my_ref = & my_orig_val;　　**D. int &my_ref = my_orig_val;**

(6) 下列哪一项不适合使用引用?

A. 为了存储一个未分配存储区中动态分配出来的地址
B. 为了避免一个较大的值传递给函数时的复制操作
C. 为了强制函数的一个参数，其值永远不能为NULL
D. 为了让一个函数能够访问传递给它的原始变量，而无需使用指针

第 14 章问答题答案

(1) 下列哪一项是C++中分配内存的正确关键字?

A. new　　B. malloc　　C. create　　D. value

(2) 下列哪一项是C++中释放内存的正确关键字? ①

A. free　　**B. delete**　　C. clear　　D. remove

(3) 以下说法哪一项是正确的?

A. 数组与指针是一样的
B. 数组不能够被赋值给指针
C. 指针可以被当做数组，但两者是不一样的

---

① 好吧，如果这两题你回答的是malloc和free，也算对，这两个是从C延续过来的函数——但你可能没有阅读本章!

D. 可以像数组一样地使用指针，但不能将数组分配给指针

(4) 下面的代码中，x、p_int和p_p_int最终的值是多少？（请注意，由于整数和指针是不同的类型，编译器不会直接接受此代码，但此练习是有用的，通过纸上的代码同样有助于理解多维指针。）

```
int x = 0;
int *p_int = & x;
int **p_p_int = & p_int;
*p_int = 12;
**p_p_int = 25;
p_int = 12;
*p_p_int = 3;
p_p_int = 27;
```

A. x = 0, p_p_int = 27, p_int = 12

B. x = 25, p_p_int = 27, p_int = 12

C. x = 25, p_p_int = 27, p_int = 3

D. x = 3, p_p_int = 27, p_int = 12

(5) 你怎样能表明一个指针没有指向有效的值？

A. 将指针置为负数

B. 将指针置为NULL

C. 释放与指针相关联的内存

D. 将指针置为false

第 15 章问答题答案

(1) 链表相比于数组的优势是？

A. 链表的每个元素占用空间较少

B. 链表可以动态地扩展新元素而不用复制已有元素

C. 链表可以更快地找到特定元素

D. 链表可以将结构体容纳为元素

(2) 以下哪项是正确的？

A. 没有任何理由使用数组

B. 链表和数组具有相同的性能特点

C. 链表和数组都允许按索引以常量时间访问元素

D. 在链表中间插入元素比在数组中间插入速度要快

(3) 通常什么时候使用链表？

16

A. 只需要存储一个元素时
B. 需要存储的元素个数在编译时刻已知时
C. 需要动态地添加和删除元素时
D. 需要快速访问已排序的列表中的任何一个元素，而无需做任何迭代时

(4) 为什么声明一个引用了自身元素类型（`struct Node { Node* p_next; };`）的链表不会有问题？

A. 这是不允许的
B. 因为编译器能够弄清楚你实际上并不需要自引用元素的内存
C. 因为该类型是一个指针，你只需要足够的空间来容纳一个指针，实际的下一个节点的内存之后才会分配
D. 只有你实际上不分配`p_next`指向另一个结构体时才允许这么做

(5) 为什么在链表末尾有一个空指针（`NULL`）很重要？

A. 它指示了链表结束的位置，防止代码访问未初始化的内存
B. 它能防止列表循环引用
C. 它能帮助调试，如果你试着偏离列表太远，程序将崩溃
D. 如果我们不存储`NULL`，那么列表将因为自引用而需要无限的内存

(6) 链表和数组的有什么相似性？

A. 两者都允许你快速地在当前列表的中间添加元素
B. 两者都允许你顺次地存储数据，并顺次访问数据
C. 两者都可以通过添加元素而变得更大
D. 两者都提供了对列表中的每个元素的快速访问

第 16 章问答题答案

(1) 下列哪种情况是尾递归？

A. 当你呼唤自己的狗时
B. 当一个函数调用自身时
C. 当一个递归函数所做的最后一件事是调用自身时
D. 当你可以将一个递归算法改写成循环算法时

(2) 何种情况下适合使用递归？

A. 当你不能使用循环来实现算法时

B. 当从子问题角度要比从循环角度能更自然地表达一个算法时
C. 永远不要，真的，它太难了
D. 在使用数组和链表时

(3) 一个递归算法需要满足什么要素？

A. 基线条件和递归调用
B. 基线条件和将问题分解成问题本身的更小版本的方式
C. 重组问题的较小版本的方式
D. 以上皆是

(4) 当基线条件不完整时，会发生什么？

A. 该算法可能提前完成
B. 编译器将检测到这个问题并报错
C. 这是没有问题的
D. 可能会发生栈溢出

第 17 章问答题答案

(1) 二叉树的主要优点是？

A. 使用指针
B. 可以存储任意数量的数据
C. 允许数据的快速查找
D. 从二叉树中删除节点很容易

(2) 什么情况下适合使用链表而不是二叉树？

A. 当你需要以某种方式存储数据，使得它可以快速查找时
B. 当你希望能访问排好序的数据元素时
C. 当你需要能够快速地将数据添加到前端或末端，但从不访问中间的元素时
D. 当你不需要释放正在使用的内存时

(3) 以下哪一项表述正确？

A. 数据添加到二叉树的顺序不同可以影响到最终树的结构
B. 应该排好序后再将数据插入到二叉树中，以便获得最佳的树结构
C. 如果元素是随机插入到二叉树中的，那么，链表查找节点的速度会比二叉树快。
D. 二叉树永远不可能退化到跟链表相同的结构

(4) 以下关于二叉树查找节点速度快的解释，哪一项是正确的？

A. 速度一点都不快，每个节点有两个指针意味着你必须做更多的事来遍历树
B. 每经过树的一层，你大约砍掉了剩余节点数量的一半
C. 二叉树并不是真的比链表好
D. 二叉树的递归调用比链表的循环遍历要快

第 18 章问答题答案

(1) 什么时候适合使用vector？

A. 当你需要存储一个键和一个值之间的关联时
B. 当你为了最大限度地提高性能而需要改变元素的集合时
C. 当你不想关心数据结构进行更新的细节时
D. 就好像面试要穿西装一样，使用vector总是没错的

(2) 怎样从一个map中一次性删除所有元素？

A. 将元素设置为空　　B. 调用`erase`方法
C. 调用`empty`方法　　**D. 调用`clear`方法**

(3) 什么时候你应该实现自己的数据结构？

A. 当你速度要求很快的时候
B. 当你需要鲁棒性强的时候
C. 当你想要利用原始数据结构的优势时，比如建立一棵表达式树
D. 你永远不需要实现自己的数据结构，除非你喜欢这么干

(4) 以下哪一项正确地声明了一个vector的迭代器？

A. `iterator<int> itr;`　　B. `vector::iterator itr;`
C. `vector<int>::iterator itr;`　　D. `vector<int>::iterator<int> itr;`

(5) 以下哪一项正在访问一个map的迭代器的键？

A. `itr.first`　　**B. `itr->first`**　　C. `itr->key`　　D. `itr.key`

(6) 怎样知道一个迭代器是否可用？

A. 跟`NULL`进行比较
B. 跟迭代的对象调用`end()`的结果进行比较
C. 检查它是否等于`0`
D. 跟迭代的对象调用`begin()`的结果进行比较

第 19 章问答题答案

(1) 下面的代码哪些是合法的?

A. `const int& x;`
B. `const int x = 3; int *p_int = & x;`
C. `const int x = 12; const int *p_int = & x;`
D. `int x = 3; const int y = x; int& z = y;`

(2) 下面的函数签名中，哪一项可以让代码`const int x = 3; fun( x );`能够编译通过?

A. `void fun (int x);`
B. `void fun (int& x);`
C. `void fun (const int& x);`
D. A和C

(3) 判断一个字符串搜索没有成功找到目标元素的最好方式是?

A. 比较结果位置和`0`
B. 比较结果位置和`-1`
C. 比较结果位置和`string::npos`
D. 检查结果位置是否大于字符串长度

(4) 如何为一个`const`的STL容器创建迭代器?

A. 声明迭代器为`const`
B. 使用索引来循环遍历，不使用迭代器
C. 使用`const_iterator`
D. 声明模板类型为`const`

第 21 章问答题答案

(1) 下列那个不属于C++构建过程中的一部分?

A. 链接
B. 编译
C. 预处理
D. 后续处理

(2) 你在什么时候会遇到一个关于未定义函数的错误?

A. 在链接的过程中
B. 在编译的过程中
C. 在程序启动时
D. 在你调用方法的时候

(3) 下列哪项会在你重复引入头文件时发生?

A. 重复声明的错误
B. 没有异常，头文件总是只会被载入一次
C. 与头文件本身是如何实现的有关
D. 头文件一次只能被一个源文件引入，所以不会出现问题

(4) 把编译和链接分开有什么好处?

A. 没有好处，这样做会让人感到迷惑并且有可能很慢因为有多个程序要运行
B. 更容易分析错误因为你可以知道问题出自链接器还是编译器
C. 这样做只让有改动的文件重新编译，节省了编译和链接的时间
D. 这样做只让有改动的文件重新编译，节省了编译时间

第 22 章问答题答案

(1) 使用函数而不直接访问数据的好处是什么？

A. 函数可以被编译器优化来提供更快的访问速度
B. 函数可以对调用者隐藏自己的实现逻辑，这样便于改变该函数的调用者
C. 使用函数是在多个源文件之间共享同一个数据结构的唯一途径
D. 没有什么好处

(2) 在什么情况下应该把代码放进一个通用的函数中呢？

A. 在你需要调用它的时候
B. 在你开始从很多地方调用同一段代码的时候
C. 在编译器开始抱怨函数太大而不能编译的时候
D. B和C

(3) 为什么要隐藏数据结构的表示方式？

A. 让数据结构更便于替换
B. 让使用该数据结构的代码更容易让人理解
C. 让代码中别的地方使用该数据结构时更容易
D. 以上都正确

第 23 章问答题答案

(1) 你为什么需要使用方法而不是直接使用结构体的字段？

A. 因为方法更加易读
B. 因为使用方法程序会更快
C. 你不应该使用方法，就应该直接使用字段
D. 这样做你可以修改数据的表现形式

(2) 下列哪个定义了与结构体`struct MyStruct { int func(); };`相关联的方法？

A. `int func() { return 1; }`

B. `MyStruct::int func() { return 1; }`

C. `int MyStruct::func() { return 1; }`

D. `int MyStruct func () { return 1; }`

(3) 你为什么想要把方法的定义内联在一个类中?

A. 这样可以让该类的使用者看到这个方法是怎么工作的

B. 因为这样会让代码跑得更快

C. 你不能这么做！这样会泄漏方法实现的细节

D. 你不能这么做，这会让程序跑得更慢

第 24 章问答题答案

(1) 为什么要使用私有数据?

A. 为了让数据更安全，免受黑客攻击

B. 为了防止其他程序员接触那些数据

C. 为了分清楚哪些数据是应该只用来实现类的

D. 你不应使用私有数据，那样会使程序更难写

(2) 类和结构体有什么不同?

A. 没什么不同

B. 类默认所有成员都是公共的

C. 类默认所有成员都是私有的

D. 类可以让你指定字段是公共的还是私有的，结构体不能

(3) 你应该怎样处理类当中的数据字段?

A. 把它们默认设为公共的

B. 把它们默认设为私有，如果有需要就移到公共的部分

C. 永远不要把它们设为公共的

D. 类通常都没有数据，但是如果有，　直接使用

(4) 你如何决定一个方法是否应该被设为公共的?

A. 永远不要把方法设为公共的

B. 一直把方法设为公共的

C. 如果方法需要使用类的主要特性就把它设为公共的，否则设为私有的

D. 如果有人可能会想要使用这个方法，那么就把它设为公共的

第 25 章问答题答案

(1) 你在什么时候需要给类写一个构造函数?

A. 总是需要写，没有构造函数你就不能使用这个类
B. 在你需要以非默认值来初始化类的时候
C. 永远不需要，编译器总是会为你提供一个
D. 只有你同时需要一个析构函数的时候

(2) 析构函数和赋值操作符之间的关系是什么?

A. 它们没什么关系
B. 类的析构函数会在运行赋值操作符之前被调用
C. 赋值操作符需要指出哪些内存应当被析构函数删除掉
D. 赋值操作符必须确保运行被复制类的析构函数和运行新类的析构函数都是安全的

(3) 在什么时候需要使用一个初始化列表?

A. 在你想要让构造函数尽可能地高效以及想要避免构造空的对象时
B. 在你初始化一个常量时
C. 在你想要运行类的某个字段的非默认构造函数的时候
D. 上面所有的都成立

(4) 下面代码的第二行执行时哪个函数会运行?

```
string str1;
string str2 = str1;
```

A. str2的构造函数和str1的赋值操作符
B. str2的构造函数和str2的赋值操作符
C. str2的复制构造函数
D. str2的赋值操作符

（因为str2还未被初始化，复制构造函数运行，而非赋值操作符。）

(5) 下面代码中哪些函数被调用了，顺序是怎样的?

```
{
    string str1;
    string str2;
}
```

A. str1的构造函数，str2的构造函数
B. str1的析构函数，str2的构造函数

C. str1的构造函数，str2的构造函数，str1的析构函数，str2的析构函数
D. str1的构造函数，str2的构造函数，str2的析构函数，str1的析构函数

(6) 如果已知一个类有个非默认的构造函数，下列关于它的赋值操作符哪个应该是正确的？

A.它应当有个默认的赋值操作符
B.它应当有个非默认的赋值操作符
C.它应当有个声明了但是没有实现的赋值操作符
D. B 或 C 正确

（它应被设为私有，这样编译器能更早地发现问题。）

第 26 章问答题答案

(1) 父类的析构函数在什么时候运行？

A. 只有对一个指向父类的指针调用delete来销毁对象的时候
B. 在子类的析构函数被调用之前
C. 在子类的析构函数被调用之后
D. 在子类的析构函数被调用的时候

(2) 给定下列的类层级，在Cat的构造函数中你需要做什么？

```
class Mammal {
public:
    Mammal (const string& species_name);
};

class Cat : public Mammal
{
public:
    Cat();
};
```

A. 没什么特别要做的
B. 使用初始化列表来调用Mammal的构造函数同时带上参数"cat"
C. 在Cat的构造函数中调用Mammal的构造函数，带上参数"cat"
D. 你应当删除Cat的构造函数并且使用默认的版本，默认构造函数会为你解决这个问题

(3) 下面的这个类定义哪里错了？

```
class Nameable
{
    virtual string getName();
};
```

A. 它没有把getName方法设为公有
B. 它没有虚的析构函数
C. 它没有getName方法的实现，但是又没有把getName声明为纯虚的
D. 上面说得都对

(4) 当你在一个接口类中声明一个虚方法时，另外的一个函数需要怎么做，才可以使用这个接口方法在子类上调用方法?

A. 把接口当做一个指针参数（或者一个引用参数）
B. 什么都不用做，它会自动复制对象
C. 它需要知道被调用方法所在子类的名字
D. 我迷惑了！虚方法是什么?

(5) 继承是如何改善重用的?

A. 通过允许代码从父类继承方法
B. 通过允许父类为子类实现虚方法
C. 通过允许代码期待接收一个接口，而不是一个具体的类，允许新的类来实现接口同时保持旧的代码可用
D. 通过允许新的类继承一个具体的类的特性，这些特性可以为虚方法所使用

(6) 下列关于类的访问级别哪个是正确的?

A. 子类只能访问父类的public方法和数据
B. 子类能够访问父类的private方法和数据
C. 子类只能访问父类的 protected 方法和数据
D. 子类可以访问父类的protected或者public方法和数据

第 27 章问答题答案

(1) 什么情况下需要使用using namespace指令?

A. 在所有头文件中，紧跟着include指令后面
B. 根本不能用，它们是危险的
C. 可以用在任何没有命名空间冲突的cpp文件顶端
D. 在你使用来自那个命名空间的变量之前

(2) 我们何以需要命名空间呢?

A. 为了给编译器的作者增加一些有趣的工作
B. 为代码提供更好的封装

C. 为了阻止大规模代码库中的命名冲突
D. 为了帮助阐明一个类的作用

(3) 在什么情况下应当把代码放到命名空间中?

A. 代码什么时候都应该放在命名空间中
B. 当你在开发一个有超过数十个文件的大规模程序的时候
C. 在你开发一个用来与别人共享的函数库的时候
D. B和C都正确

(4) 为什么不能把使用命名空间的声明放在头文件中?

A. 这么做是非法的
B. 没有理由不放在头文件中，使用的声明只有在头文件中才是合法的
C. 这么做会把使用声明强加给任何包含了这个头文件的人，即使这样会导致冲突
D. 如果多个头文件包含了使用声明就会导致冲突

第 28 章问答题答案

(1) 哪个数据类型可以用来读取文件?

A. `ifstream`　B. `ofstream`　C. `fstream`　**D. A和C**

(2) 下列哪句是正确的?

A. 文本文件比二进制文件占用更少的内存空间
B. 二进制文件更易于调试
C. 二进制文件比文本文件更节省空间
D. 文本文件太慢了，不能在真实的程序中使用

(3) 在写入二进制文件的时候，为什么不能传入一个指向字符串对象的指针?

A. 你每次都要传入一个`char*`到`write`方法中
B. 内存中可能没有保存字符串对象
C. 我们不知道字符串对象的布局，它可能含有会被写入到文件中的指针
D. 字符串太大了必须一点一点地写入

(4) 下列关于文件格式哪个是正确的?

A. 文件格式和别的输入一样易于修改
B. 修改文件格式需要考虑旧版本的程序读取文件时会发生什么事

C. 设置文件格式时需要考虑新版本的程序打开旧版本的文件会发生什么事
D. B和C

第 29 章问答题答案

(1) 什么时候应该使用模板？

A. 你想要节省时间的时候
B. 你想要代码运行地更快的时候
C. 当你需要为不同的类型多次写同样的代码的时候
D. 当你需要确保之后可以重用代码的时候

(2) 你什么时候需要为模板参数提供一个类型？

A. 总是需要
B. 只有在声明一个模板类的实例的时候
C. 只有类型无法推断出来的时候
D. 对于模板函数，只有在类型无法推断的时候，对于模板类，一直都需要

(3) 编译器如何辨别一个模板参数可以用于一个给定的模板？

A. 它会实现一个特定的C++接口
B. 声明模板的时候你必须指定约束条件
C. 它会尝试使用模板参数；如果参数类型支持所有需要的操作，编译器就会接受它
D. 在声明模板的时候你必须列出所有合法的模板类型

(4) 把模板类放在头文件中和把一个常规的类放在头文件中有什么不同？

A. 没什么区别
B. 常规类不能在头文件中定义它的任何方法
C. 模板类必须把所有的方法在头文件中定义
D. 模板类不需要有对应的`.cpp`文件，但是常规的类需要有

(5) 什么时候应该把函数写成模板函数？

A. 一开始的时候——你永远不会知道什么时候需要对不同的类型使用相同的逻辑，所以你应当总是写模板方法
B. 只有在你无法把所给的类型转换成函数当前需要的类型的时候
C. 当你写了几乎一样的逻辑，但是处理的是一个与第一个函数所使用的类型有着相似特性的不同类型时

D. 当两个函数做“几乎是”相同的事，而且你可以通过几个额外的`Boolean`参数就把逻辑修改过来时

(6) 你什么时候会知道所写的模板存在的大部分错误？

A. 在你编译模板的时候
B. 在链接的阶段
C. 在你运行代码的时候
D. 在你第一次编译初始化模板的代码时

索 引

M

N

P

Q

R

S

W

X

Y

Z

亚马逊读者评论

“作为一个多年来习惯于Fortran编程的C++编程新手，我认为本书内容简明扼要。读者只需从网上下载软件，然后即可按自己的节奏完成书中练习。”

“我对C有一定的了解，因此理所当然地会很容易看懂本书内容。但是，当我真正开始阅读它，才惊喜地发现，它让我对编程有了重新的认识，真正充分理解了C与C++编程。”

“我是一个IT培训师，之前针对C++为儿子买了3本书，而目前为止我认为本书是组织结构最好的一本学习用书。我有30多年学习C的经历。本书从常见的语言特性讲起，简单流畅地过渡到对象，并介绍了鲜见于其他同类图书的模板、文件I/O、格式化I/O和命令行参数等内容。”

“本书并非是内容全面的大部头著作，但其内容阐述真的很棒！……指针是语言中非常晦涩难懂的部分，也是我之前常常存有困惑的内容，而本书对指针的介绍堪称完美。……无论你之前是否具有编程经验，本书绝对对你大有帮助！”

- 作者两度荣获哈佛大学Top Teaching Fellow
- 数百万月访问量C\C++教程网站提供支持
- 体现C++编程的现代观点，有效解决实际问题

本书不是一本百科全书式的C++教程，更不是一本写给有编程经验的人看的C++书。如果你想学C++，但没有太多编程经验，而且十分发愁去看那些厚得要命，大部分内容不知所云，更不知道何年何月才用得上的C++“砖头书”，别着急，就看这本吧！

本书篇幅适中，写得又简单通俗，涵盖了C++编程的所有重要概念。另外，我们得提一提本书作者Alexander Allain，他是月访问量超百万的著名C\C++教程站Cprogramming.com的创建者，拥有在哈佛大学讲授C++编程的一线教学经验。本书就是他结合多年教学心得和大量读者反馈，为普通C++初学者登堂入室特意编写的一本全新教程，可以帮助你迅速成长为一名优秀的、受欢迎的C++程序员。

作者真正了解每一位C++编程学习者的需求，了解初学者起步阶段的困惑和纠结。因此，本书由浅入深、循序渐进、步步为营，讲述了编程过程的每一个环节，揭示了编程之路中可能遇到的各种“坑”。以下内容是本书特有的教学思想和方法的体现。

- 从编程所需的工具开始讲起，耐心教你怎么使用
- 清晰解释变量、循环、函数等最基本的编程概念
- 手把手示范怎么把头脑中的想法转换成C++代码
- C++的指针不好理解，但本书会给你最清晰明白的解释
- 字符串、文件I/O、数字、引用……纷至沓来
- C++中的类，以及类的设计
- 面向C++的特有编程模式
- 使用C++进行面向对象编程
- 数据结构和标准模板库（STL）
- 习题和75个课后练习巩固你对重要概念和知识点的理解

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/程序设计/C++

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-35700-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks